VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

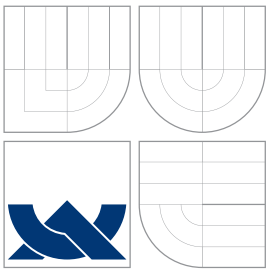
UŽIVATELSKÉ ROZHRANÍ PRO EVIDENCI A ŘÍZENÍ NÁVŠTĚV

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

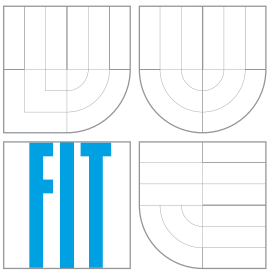
AUTOR PRÁCE
AUTHOR

PETR NOSEK

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

UŽIVATELSKÉ ROZHRANÍ PRO EVIDENCI A ŘÍZENÍ NÁVŠTĚV

USER INTERFACE FOR MONITORING AND CONTROL OF VISITORS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETR NOSEK

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Dr. Ing. ZEMČÍK

BRNO 2009

Abstrakt

Tato bakalářská práce obsahuje popis návrhu a implementace elektronického uživatelského rozhraní pro evidenci a řízení návštěv. V této eletronické evidenci jsou zaznamenávány informace o příchozích návštěvách a přidělených návštěvnických kartách. S návštěvnickou kartou se osoba může volně pohybovat po objektu. Recepční má neustále přehled o aktuálních návštěvách a jejich době pobytu. V práci jsou také zmíněny techniky a postupy, které vedly k urychlení vývoje a snížení výskytu chyb při vývoji.

Klíčová slova

grafické uživatelské rozhraní, evidence a řízení návštěv, NUnit, NHibernate, LINQ, log4net, Object Relational Mapping, Mono, LINQ to NHibernate

Abstract

This document contains a description of a design and implementation of a user interface for monitoring and control of visitors. Information about incoming visitors and assigned guest cards are recorded in this electronic evidence. Guest card allows a user free movement inside the object. Receptionist has always a summary of current visitors and the length of their stays. In the document there are also mentioned techniques and procedures which speeded up program development and reduced number of errors.

Keywords

graphical user interface, monitoring and control of visitors, NUnit, NHibernate, LINQ, log4net, Object Relational Mapping, Mono, LINQ to NHibernate

Citace

Petr Nosek: Uživatelské rozhraní pro evidenci a řízení návštěv, bakalářská práce, Brno, FIT VUT v Brně, 2009

Uživatelské rozhraní pro evidenci a řízení návštěv

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Doc. Dr. Ing. Pavla Zemčíka.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem při vypracování čerpal.

. . . . . .
Petr Nosek
19. května 2009

Poděkování

Chtěl bych poděkovat panu Doc. Dr. Ing. Pavlu Zemčíkovi a Ing. Oldřichu Sojovi za cenné rady při zpracování této práce a za jejich čas, který mi věnovali.

Obsah

Kapitola 1

Úvod

Počítače jsou obsaženy ve všech sférách našeho života. Pominuly doby, kdy stály na vrátnici píchací hodiny a každý zaměstnanec si při vstupu do podniku ručně zaznamenal čas příchodu. Metoda ručního značení nám nijak neusnadňuje kontrolu přítomnosti zaměstnanců a už vůbec ne výpočet mzdy na základě docházky. Právě v tuto chvíli přicházejí elektronická zařízení, aby nám usnadnila evidenci docházky.

Stejně jako rostly požadavky na zaznamenávání údajů o zaměstnancích, rostly i požadavky zaznamenávání příchodů návštěv. S vedením papírové knihy návštěv si vystačíme pouze ve velmi malých podnicích, čítajících maximálně dvacet zaměstnanců. Jak si ale poradíme s návštěvní knihou v podniku s více jak 500 zaměstnanci? Co když budeme potřebovat zjistit, zda je navštěvovaný zaměstnanec přítomen v podniku? Jakým způsobem rychle vyhledat, kolik je aktuálně přítomných návštěv nebo kolik návštěv přišlo v daném časovém rozmezí? Všechny tyto otázky řeší elektronické zpracování, chceme-li elektronická evidence cizích osob.

Představme si vrátnou obsluhující počítač. Při příchodu návštěvy vloží do počítače její jméno a návštěva obdrží návštěvnickou kartu. Vhodně zvolenou návštěvnickou kartou může být omezen pohyb návštěvy po objektu.

V následujících kapitolách této práce je popsáno uživatelské rozhraní pro evidenci a řízení návštěv jak z hlediska návrhu, tak samotné implementace podle konkrétního zadání.

Ve druhé kapitole se čtenář seznámí s aktuálně dostupným softwarem pro evidenci a řízení návštěv. Čtenář tak bude uveden do problematiky, získá základní informace o funkcích, jaké tyto systémy nabízí, jaké nabízí výstupy a jaké jsou požadavky provozu.

Třetí kapitola představuje základní cíle projektu. V návaznosti na druhou kapitolu přidává přehled všeobecných požadavků pro systémy evidence návštěv. Vysvětluje možnosti ve volbě implementační platformy a v neposlední řadě představuje technologie, které dopomohou k dosažení stanovených cílů.

Čtvrtá kapitola se zabývá samotnou implementací. Popisuje smysl a účel jednotlivých knihoven výsledného produktu a představuje celkové řešení grafického rozhraní.

V závěru je celkově zhodnocen výsledek z hlediska dosažení cílů, kvality a budoucí perspektivy tohoto uživatelského rozhraní.

Na úplném konci této práce je připojen seznam literatury, ze kterého bylo čerpáno a také je zde uveden seznam příloh, které jsou její součástí. Za účelem rychlé orientace v tomto dokumentu byl vytvořen rejstřík.

Kapitola 2

Současný stav

V současné době existuje na trhu nepřeberné množství uživatelských rozhraní pro evidenci a řízení návštěv. Tato uživatelská rozhraní jsou ve většině případů součástí velkých celků, tzv. docházkových systémů. Evidence a řízení návštěv patří mezi volitelné moduly těchto docházkových systémů. Účelem této kapitoly je ukázat přehled docházkových řešení, která jsou v současnosti dostupná na trhu a ukázat existující systémy pro správu verzí a chyb.

Za účelem lepší orientace a porovnávání docházkových systémů mezi sebou, je kapitola zaměřena na tyto dílčí části.

1. Databáze

   - s architekturou typu „PC file-server“:
     Souborový přístup - data jsou uložena v souborech, klient přistupuje k souborům. Výhodou je jednoduchost řešení (odpadá instalace databázového serveru). Hlavní nevýhodou je nespolehlivost síťového řešení. Systém je spolehlivý pouze v případě, že klient i databázový server běží na jednom počítači. Mezi zástupce těchto databázových řešení patří například dBASE, FoxBase, aj.
   - s architekturou typu „klient-server“:
     Oproti předchozímu typu databáze lze považovat za nevýhodu složitější instalaci. Výhodou pak bude spolehlivost síťového řešení. Tuto architekturu je vhodné nasadit v prostředí, kde očekáváme více klientských PC přistupujících přes síť k databázi. Mezi příklady databázových serverů tohoto typu se řadí například MS SQL, MySQL, Firebird aj.

2. Docházkový server
   Komunikuje se čtecími terminály rozmístěnými po objektu (budově). Čtecí zařízení uloží v paměti surová data a docházkový server ověřuje jejich validitu, provádí přepočty aj. Načtená data jsou poté vložena do databáze. Server také umožňuje konfiguraci čtecích terminálů v objektu (nastavení povolení přístupu).

3. Klient
   Tenký klient. Vykonává minimum aplikační logiky.

   - dvouvrstvá architektura
     Klient načítá data přímo z databázového serveru.
   - třívrstvá architektura
     Klient načítá data přímo z docházkového serveru. Docházkový server je prostředníkem, který zprostředkovává data z databázového serveru.
   - dvouvrstvá architektura s využitím docházkového serveru
     Klient načítá data přímo z databázového serveru a pro konfiguraci čtecích zařízení využívá docházkový server.

2.1 Rozdělení

Komerční systémy, popisované níže, převážně používají databáze s architekturou typu klient-server kombinovaně s klientem s dvouvrstvou architekturou s využitím docházkového serveru. Objevují se i nabídky s databázovou architekturou file-server s doporučením pro velmi malé firmy.

Jelikož jsou uživatelská rozhraní pro evidenci a řízení návštěv určena především na recepce středně velkých a velkých firem, nemá smysl již dále uvažovat databáze s architekturou typu file-server.

Na poli docházkových systémů je silné konkurenční prostředí, a proto jsem vybral reprezentační vzorek firem, věnujících se docházkovým systémům.

Softwarové řešení mezi sebou nelze jednoznačně srovnávat, protože každá firma se zaměřuje na jiný segment trhu. Přesto jsem se pokusil údaje o produktu rozřadit do čtyř kategorií.

1. Vstupní údaje při příchodu návštěvy
2. Hlavní funkce programu
3. Přehledy
4. Požadavky

2.2 Z-WARE

Společnost Z-WARE[1] se zabývá docházkovými, stravovacími, přístupovými a domovními systémy. Jeden z modulů docházkového systému tvoří právě modul návštěv.

Pracovník recepce zaznamená příchozí návštěvu, při jejím odchodu zaznamená ukončení návštěvy. Uživatelské rozhraní umožňuje okamžitý přehled o počtu návštěv v určitém období. Přehled návštěv může být přístupný všem uživatelům.

- Vstupní údaje při příchodu návštěvy:
  - datum a čas příchodu - systém nabídne aktuální čas, recepční má možnost zadat jinou hodnotu
  - jméno a firma návštěvníka - lze zapsat novou osobu nebo vybrat ze seznamu již dříve evidovaných návštěv
  - počet osob - skupinová návštěva
  - SPZ automobilu - SPZ je vázána na konkrétní návštěvu. Software neřeší případného spolujezdce.
  - navštívená osoba nebo jiný cíl návštěvy
- Hlavní funkce programu:
  - evidované návštěvy jsou řazeny od nejnovější po nejstarší - zobrazení je na hlavní obrazovce programu.
  - neukončené návštěvy jsou zvýrazněny jinou barvou pozadí
  - v horní části okna jsou uvedeny základní přehledy (součty osob/automobilů, přítomných lidí ve firmě)
  - zobrazené údaje lze editovat
  - lze kdykoli odepsat/zablokovat zvolenou návštěvu
  - ukončení návštěvy probíhá tak, že systém nabídne seznam dosud neukončených návštěv a recepční potvrzuje výběr

1. Další informace o firmě dostupné z `http://www.z-ware.cz`

- Přehledy:
  - výběr období, v rámci kterého bude proveden výstup
  - fitrace podle jména návštěvy (možno pouze část jména)
  - filtrace podle názvu firmy (možno pouze část názvu)
  - filtrace podle telefonního čísla navštívené osoby (možno pouze část čísla)
- Požadavky:

druh	**softwarové požadavky**
docházkový server	OS Windows
klient	OS Windows 98 a vyšší
databáze	MS SQL, MySQL, PostgreeSQL, Oracle 9

Tabulka 2.1: Z-WARE požadavky

2.3 RON

Společnost RON[2] bohužel disponuje pouze malým množstvím informací, nicméně i tato firma stojí za zmínku.

Součástí docházkového systému je modul VISITOR - rozhraní pro evidenci návštěv.

- Vstupní údaje při příchodu návštěvy:
  - datum a čas příchodu
  - jméno návštěvníka
  - navštívená osoba nebo jiný cíl návštěvy
- Hlavní funkce programu:
  - modul VISITOR je připraven na spolupráci se čtecím zařízením dokladů (např. občanský průkaz)
  - výhodou docházkového systému je shopnost spolupracovat se mzdovými a personálními systémy
- Požadavky:

druh	**softwarové požadavky**
docházkový server	OS Windows
klient	OS Windows 98 a vyšší
databáze	MS SQL 2000, MS SQL 2005, Firebird, Oracle 9,10, Sybase

Tabulka 2.2: RON požadavky

2.4 ANeT

Společnost ANeT[3] nabízí produkty různých velikostí a specializovaná řešení. Mezi jejich hlavní produkty patří docházkové systémy, systémy pro plánování lidských zdrojů, přístupové systémy, stravovací systémy a vrátnice, která plní funkci evidence návštěv.

2. Další informace o firmě dostupné z `http://www.ron.cz`
3. Další informace o firmě dostupné z `http://www.anet.info`

- Vstupní údaje při příchodu návštěvy:
  - datum a čas příchodu
  - jméno a firma návštěvníka - lze zapsat novou osobu nebo vybrat ze seznamu již dříve evidovaných návštěv
  - automobil - software řeší evidenci automobilů a umožňuje provázat je s konkrétními návštěvami
  - navštívená osoba
  - účel návštěvy
  - trvání návštěvy
  - informaci o vrátné, která kartu vydala a přijala zpět (systém vkládá automaticky, předpokládá se více vrátnic)
- Hlavní funkce programu:
  - průběžné sledování průchodů vlastních zaměstnanců se zobrazením fotografie
  - zobrazené údaje lze editovat
  - možnost vložení osoby na tzv. černou listinu a zakázat jí vstup
  - zjištění přítomnosti navštěvovaných osob - pokud přichází návštěva za konkrétní osobou, je vhodné zjistit, přítomnost konkrétní osoby
  - zobrazuje přítomnost zaměstnanců v jednom středisku nebo v celé společnosti a to ke konkrétnímu datu. Možnost omezení zobrazení pouze na přítomné, pouze nepřítomné nebo obojí najednou.
  - zadání průchodu pracovníkovi, který si zapomněl identifikační kartu (v tomto případě se uvede průchozí terminál PC)
  - nabízí propojení s kamerami
- Přehledy:
  - výběr období, v rámci kterého bude proveden výstup - až 23 hodin zpětně
  - na záznamy lze aplikovat různé omezující filtry - pro konkrétní výběr
- Požadavky:

druh	**softwarové požadavky**
docházkový server	OS Windows 2000 a vyšší
klient	OS Windows 2000 a vyšší
databáze	MS SQL, Oracle

Tabulka 2.3: ANeT požadavky

2.5 COMINFO

Společnost COMINFO [4] nabízí systém INFOS, který představuje ucelený balík softwarových aplikací od řízení a definování vstupů, přes zpracování docházky, evidenci návštěvníků, ovládání parkovišť, řízení výtahů až po řešení objednávkového i restauračního stravování. Modul pro evidenci návštěv nese název VISIT.

4. Další informace o firmě dostupné z `http://www.cominfo.cz`

- Vstupní údaje při příchodu návštěvy:
  - datum a čas příchodu
  - jméno a firma návštěvníka - lze zapsat novou osobu nebo vybrat ze seznamu již dříve evidovaných návštěv
  - navštívená osoba nebo jiný cíl návštěvy
  - evidence vozidel
- Hlavní funkce programu:
  - zobrazené údaje lze editovat
  - lze kdykoli odepsat/zablokovat zvolenou návštěvu
  - nabízí možnost kdykoli odepsat kartu návštěvy
- Přehledy:
  - výběr období, v rámci kterého bude proveden výstup
  - fitrace podle potřeby
  - dávkové (měsíční) uložení dat do archivu
- Požadavky:

druh	**softwarové požadavky**
docházkový server	OS Windows
klient	OS Windows 98 a vyšší
databáze	MS SQL 2000, Oracle, Informix

Tabulka 2.4: COMINFO požadavky

2.6 Trade FIDES

Společnost Trade FIDES[5] uvádí pouze základní informace o svém systému, za zmínku však stojí především díky zaměření na bezpečnost. Trade FIDES je také zajímavý řešením úložiště dat. Místo využití SQL databáze, vytváří vlastní produkt LATIS SQL, který zajišťuje integraci a monitoring všech technologií. Umožňuje správu z jednoho pracoviště, na kterém jsou všechna data.

Trade FIDES implementuje modul MONITOR, který eviduje přítomnost všech uživatelů na pracovišti i návštěv. MONITOR je vhodný právě pro využití na recepcích.

- Hlavní funkce programu:
  - součástí instalace je čtecí zařízení OCR kódu - umožňuje přečíst informace z občanského průkazu
  - kartě návštěvníka může být přidělena trasa, po které se může v objektu pohybovat
  - karta se deaktivuje odchozí čtečkou

5. Další informace o firmě dostupné z `http://www.fides.cz`

- Požadavky:

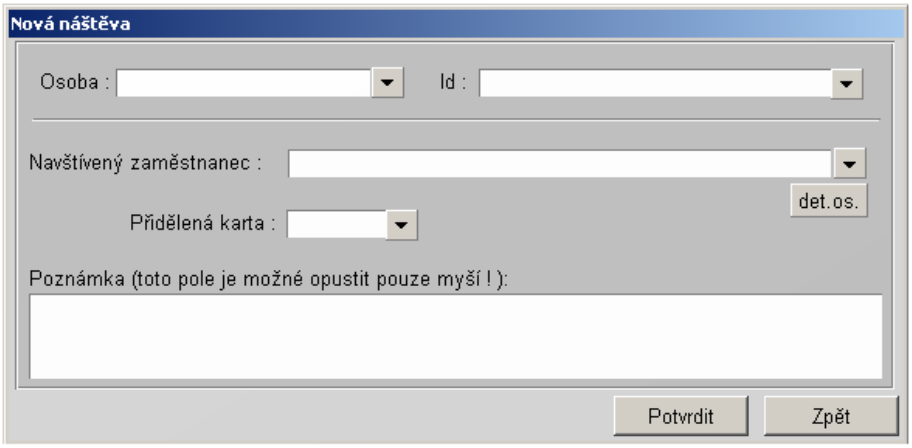
druh	softwarové požadavky
docházkový server	OS Windows
klient	OS Windows NT, 2000 a vyšší
databáze	vlastní LATIS SQL řešení

Tabulka 2.5: Trade FIDES požadavky

2.7 GOLDCARD - GCS7800

Systém GCS7800 je softwarové řešení společnosti GOLDCARD[6], které se skládá z přístupového, docházkového, stravovacího, pokladního, hotelového zámkového systému a modulem pro evidenci návštěv. Modul pro evidenci návštěv je napsán v Corel Paradox[7] a využívá databázovou achitekturu typu file-server (paradoxová databáze).

- Vstupní údaje při příchodu návštěvy:
  - datum a čas příchodu - systém nabídne aktuální čas, recepční má možnost zadat jinou hodnotu
  - jméno a firma návštěvníka - lze zapsat novou osobu nebo vybrat ze seznamu již dříve evidovaných návštěv
  - navštívená osoba nebo jiný cíl návštěvy

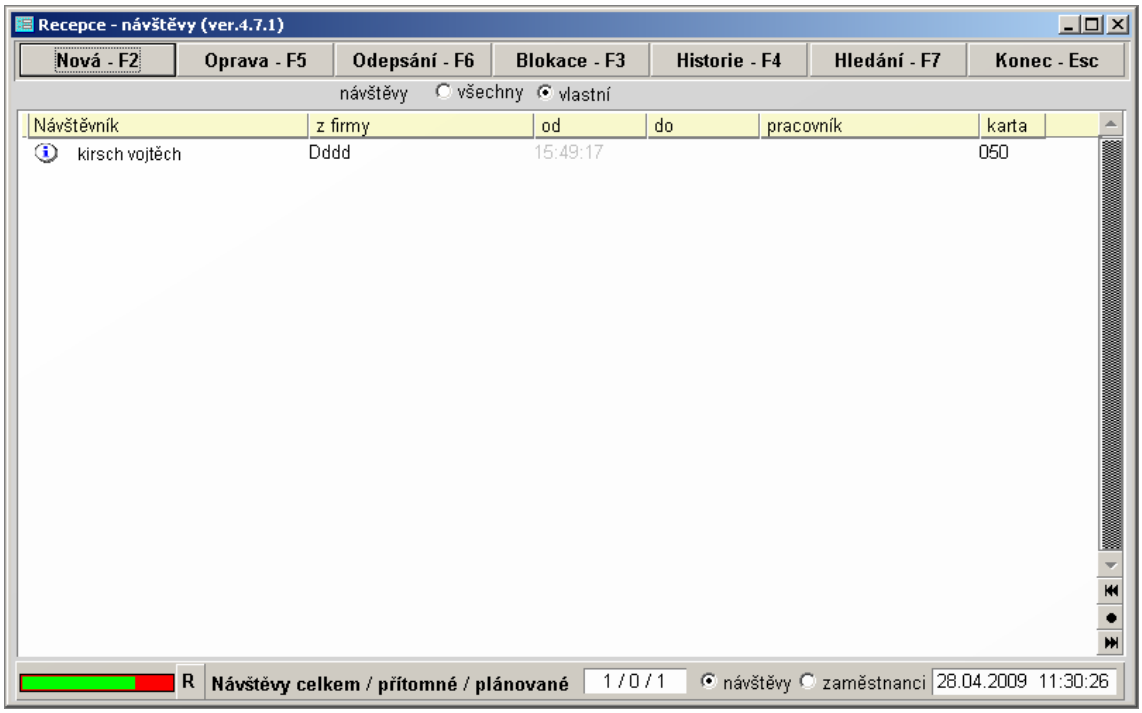


Obrázek 2.1: Okno pro vložení příchozí návštěvy

- Hlavní funkce programu:
  - evidované návštěvy jsou řazeny od nejnovější po nejstarší - zobrazení je na hlavní obrazovce programu.
  - v dolní části okna jsou uvedeny základní přehledy (součty návštěv, přítomných lidí ve firmě)

6. Další informace o firmě dostupné z `http://www.goldcard.cz`
7. Paradox je relační databázový systém firmy Corel, který patří do skupiny tzv. „malých“ stolních databází, optimalizovaných pro provoz na osobních počítačích nebo pro úlohu uživatelské části klient/server. Prostředky paradoxu umožňují například vytvořit databázovou tabulku, vkládat a vybírat data z databáze, tvořit formuláře pro zobrazení a vkládání dat, tiskové sestavy a vytvářet vlastní aplikace pomocí programovacího jazyka ObjectPAL.

- zobrazené údaje lze editovat
- lze kdykoli odepsat/zablokovat zvolenou návštěvu



Obrázek 2.2: Hlavní okno programu

- Přehledy:
  - seznam aktuálních návštěv
  - historie návštěv s možností filtrace podle:
    - data příchodu a odchodu
    - části jména, příjmení, firmy nebo karty návštěvy
    - osoby, která je navštěvována
  - detail návštěvy (jméno, příjmení, firma)
- Požadavky:

druh	**softwarové požadavky**
docházkový server	OS Windows
klient	OS Windows 98 a vyšší
databáze	PARADOX - souborový přístup

Tabulka 2.6: GOLDCARD požadavky

2.8 Subversion (SVN)

Subversion je systém pro správu a verzování zdrojových kódů, je založený na principu centrálního repozitáře. Je nezávislý na operačním systému a skládá se ze dvou hlavních částí - klientské a serverové části. Jako klienta pro Subversion server jsem použil *TortoiseSVN*[8], který je určen pro operační systém Windows. Klientská část poskytuje nástroje pro práci s verzemi a komunikaci se serverovou částí. Serverová část se stará o *repository* (centrální úložiště dat).
Hlavní přednosti a vlastnosti:

- veškeré změny zvyšují verzi - například změna obsahu souboru, přejmenování, přesunutí...
- neudržuje verze každého souboru zvlášť. Udržuje jedno číslo verze pro celou repository.
- usnadňuje práci při současném vývoji více programátorů.
- automaticky je vytvářena dokumentace, co se s jakým souborem dělo.
- kdykoli je možné se vrátit ke kterékoli verzi. Pokud tedy víme, že některá vlastnost před týdnem fungovala, není nic jednoduššího než přepnout na verzi, která proběhla před týdnem a porovnat změny.

2.9 Bugzilla

Bugzilla[9] je webová aplikace pro sledování chyb (*bug tracking*), původně vyvinutá a používaná organizací Mozilla. Jde o open source software s licencí Mozilla Public License, která je používána v mnoha proprietárních i open source projektech. [24]
Hlavní přednosti a vlastnosti:

- umožňuje sdílení informací napříč celým týmem.
- okamžitý přehled o stavu software.
- kvalifikované rozhodování o vydání verze.
- automaticky vznikající záznam historie.
- u chyb je možné definovat řadu parametrů (popis, stav, závažnost, apod.) a lze k nim přidávat komentáře, záplaty atd.
- umožňuje sledování prostřednictvím elektronické pošty (informace o provedených změnách přijdou do e-mailové schránky).
- obsahuje systém skupin a přidělování práv - různí uživatelé mohou mít různá oprávnění, nahlášené bugy mohou být viditelné pouze pro vybranou skupinu vývojářů.
- zlepšení kvality software.
- zajištění odpovědnosti.
- zlepšení komunikace v týmu.

Doplňující informace o účelu bug tracking systému jsou v použité literatuře [10], [8].

8. Dostupné z `http://tortoisesvn.tigris.org`
9. Dostupné z `http://www.bugzilla.org`

Kapitola 3

Návrh implementace

Kapitola blíže specifikuje cíle projektu a prostředky k jeho dosažení. Popisuje použité knihovny, nástroje a výhody, které tyto knihovny či nástroje do projektu přináší. Před samotným začátkem programování jsem věnoval hodně času přípravě, abych předešel případným nekompatibilitám a dal projektu maximální možnost rozvoje. Všechny mé postřehy k přípravě, jsou zaznamenány zde.

3.1 Základní cíle projektu

Na začátku jsem s potenciálním zájemcem o produkt konzultoval, co by očekával od nového softwaru. Chtěl jsem se poučit z chyb, které byly učiněny v minulosti. Zároveň jsem si kladl otázky: „Jak by to šlo udělat lépe? Jak zachovat přehlednost ve složitém projektu? Jaké jsou požadavky uživatelů? Jaké jsou problémy při vývoji současného software?“

Po konzultaci jsem došel k názoru, že můj úkol není pouhé naprogramování kódu. **Můj úkol byl mnohem komplexnější, než pouhá „implementace grafického rozhraní“.**

Z rozhovoru jsem stanovil pět hlavních cílů, na které bude od samotného začátku kladen velký důraz. Zároveň tak byl stanoven směr, kterým se bude program ubírat.

1. Evidence a řízení návštěv je pouze jeden modul, který se v budoucnu stane součástí daleko většího balíku. Evidence a řízení návštěv **musí položit základy pro další programový rozvoj**. Od začátku je třeba myslet na to, aby šlo co nejvíce kódu využít (recyklovat) v dalších modulech (které nejsou součástí této bakalářské práce). Na budoucím rozvoji se tak mohou podílet jiní programátoři.

2. **Stabilita databáze**.
   Dosavadní systém byl založen na databázi typu „file-server“, docházelo tak ve větších sítích k samovolnému poškozování tabulek. Proto musí být nasazeno jiné řešení.

3. **Přehlednost a složitost ovládání softwaru.**
   Při konzultaci vyplynula na povrch stížnost, že nemalá část problémů je způsobena nepochopením zákazníka. Zákazník dělá obslužné chyby při ovládání programu. K tomuto se nabízí hned několik otázek. Není stávající ovládání softwaru příliš složité? Je ovládání dostatečně intuitivní. Byl zákazník dostatečně zaškolen? Nikdo si bohužel nevedl statistiky četnosti a druhu chyb při ovládání programu. Bylo to však vnímáno jako dostatečně důležitý problém.
   Jak tento problém vyřešit? Obávám se, že na úplné řešení nemám dostatek vstupních informací. Mohu se ovšem postarat o **prevenci a analýzu**. Prevence spočívá v tom, že bude důsledně zaznamenáváno, co ve kterou chvíli zákazník dělá. Bude tak možné *sestavit sled událostí*, které vedly k pádu programu. Stejně tak bude možné ze záznamu vytvořit statistiky a vyhodnotit tak problémové prvky programu, což je ale dlouhodobější záležitost.
   Další krok, kterým mohu napomoci k prevenci uživatelských chyb, je zaměření se na *jednoduchost ovládání*. Důležité je vhodně a logicky seskupit ovládací prvky programu tak, aby se neznalý uživatel jednoduše dovtípil, co má pod daným ovládacím prvkem nalézt. Platí pravidlo, že *jeden obrázek vydá za 1000 slov*, proto musí být součástí programu vhodné ikony. Není třeba zákazníka ohromit velkou spoustou tlačítek, z nichž většina bude plnit

stejnou funkci, jen trochu jinak. Uživatel bude ve skutečnosti stejně využívat pouze část funkcionality programu. Především je důležité pokrýt hlavní funkce programu.
„Recyklace" formulářových oken je také krokem k přehlednosti programu. Jednoduchý příklad nalezneme při vkládání a upravování záznamů. Pokud použiji pro obě funkce stejné formulářové okno, uživatel bude okamžitě vědět, co dělat. Kdybych použil jiné okno, zbytečně udělám systém složitější a uživatel se bude muset naučit ovládat každé okno zvlášť. Na recyklaci oken je třeba myslet i do budoucna. Stejná formulářová okna v rámci celého programového celku (pro všechny moduly) jsou velmi důležitá.

4. **Chyby v programu.**
Přítomnost testera u projektu, který po vydání nové verze programu otestuje funkčnost, je naprosto **nedostačující**. Tester samozřejmě některé chyby v programu odhalí - zejména chyby v grafickém rozhraní. V testerových silách však není odhalit všechny chyby v datové části. Další nevýhodou je, že spousta organizací věnuje testování programů pouze mizivé procento času. Pokud provedeme drobnou změnu programu, nebudeme přece testera vůbec obtěžovat. Bohužel i drobná změna programu může ovlivnit (a ve většině případů také ovlivní) další funkce programu.
Není možné, aby tester všechny tyto zdroje chyb eliminoval. Lze s tím něco udělat? Ano, lze. Je načase se seznámit s pojmem *„unit testing"* a začlenit unitové testy do programu.
Pod pojem unit testing se zahrnují nástroje, metodika a činnost, jejímž cílem je ověřování správné funkčnosti dílčích částí neboli jednotek zdrojového textu.[29] Více se unit testingem budu zabývat hlouběji v této kapitole.

5. **Organizace vývoje.**
U instalace stávajícího softwaru GCS7900 docházelo v řadě případů ke kuriozním situacím. Jedním slovem chaos. Neexistoval žádný centrální systém hlášení chyb ani řádný verzovací systém. Každý servisní pracovník měl na svém PC aplikaci, kterou instaloval. Pokud v aplikaci objevil servisní pracovník chybu, vývojář dodal konkrétnímu servisnímu pracovníkovi opravenou knihovnu. Servisní pracovník pak provedl instalaci u zákazníka. V praxi to znamenalo, že každý servisní pracovník i zákazník měli unikátní verzi aplikace. Samozřejmě, že postupně existovaly snahy o vylepšení této situace, nicméně výsledek nebyl dostačující.
Proto jsem nastudoval a navrhl dvě aplikace, které zlepšily organizaci vývoje. Pro řízení verzí jsem aplikoval *Subversion* [1] a pro *hlášení chyb* (dále **bugreporting**) jsem nainstaloval bugreporting systém *Bugzilla* [2]. Obě zmiňované aplikace jsem začal aktivně používat a seznámil s nimi i servisní pracovníky.

3.2 Všeobecné požadavky na systémy evidence návštěv

Všeobecné požadavky na systémy evidence návštěv lze rozdělit do tří kategorií.

1. **Operační systém** - všechny analyzované systémy evidence návštěv pracují pouze na operačním systému Microsoft Windows. Je to dáno především kvůli orientaci na koncového zákazníka a určitě i historickým vývojem. Žádný ze systémů nebyl optimalizován pro GNU Linux.

   Nový systém evidence návštěv proto musí bezpodmínečně běžet na OS Windows. Pokud by se program podařilo optimalizovat i pro GNU Linux, byla by to sladká třešnička na dortu. Nicméně optimalizace pro GNU Linux není podmínkou.

2. **Databáze** - databázový systém musí být především stabilní a spolehlivý. Protože je modul návštěv součástí většího softwarového celku, musíme počítat s nemalým zatížením databázového serveru. Z tohoto důvodu není možné použít databázový server s architekturou

1. Dostupné z `http://subversion.tigris.org`
2. Dostupné z `http://www.bugzilla.org`

typu „PC file-server“. Z architektury typu „klient-server“ lze zvolit některou z těchto databází: MS SQL, MySQL, Firebird, Oracle.

Můžeme zvolit i některou z méně známých kvalitních databází. Nicméně vystavujeme se tak riziku, že potenciální zákazník odmítne méně známou databázi, protože má už zakoupenu některou z výše jmenovaných. Investice do správy další databáze budou proto nežádoucí.

3. **Základní funkce systému evidence návštěv** - mezi základní funkce samozřejmě patří vkládání návštěv. Vhodná je také editace návštěvy (v případě překlepu). Program musí v každém okamžiku zobrazovat aktuální stav návštěv, počet přítomných návštěv, počet ukončených návštěv, čas příchodu a čas odchodu návštěvy. V případě ztráty návštěvnické karty musí být možné zapsanou návštěvu ručně ukončit.

   Co se týče historie posledních návštěv - je nutné, aby program umožňoval filtrování uložených záznamů. Nabízí se filtrace podle data, jména návštěvy, navštěvované osoby či návštěvnické karty. Samozřejmostí je pak detail informací o návštěvě samotné, včetně celkové doby strávené v budově.

   Každého určitě napadne spousta dalších funkcí, které by modul návštěv mohl zahrnovat. Zmiňované funkce jsou pouze základní - tedy funkce, které by v žádném modulu návštěv neměly chybět.

3.3 Implementační platformy

Vhodná volba implementační platformy je důležitým krokem pro budoucí rozvoj programu. Nabízí se některý z třídních jazyků, založených na objektově orientovaném paradigmatu. Objektově orientované jazyky přináší výhody především ve vyšší míře abstrakce, jednodušší dekompozici problémů, udržovatelnosti a rozšiřitelnosti. Mají složitější sémantiku, vyžadují více času na naučení a výsledný generovaný kód je pomalejší, nicméně výhody mnohonásobně převyšují nevýhody. Jako hlavní zástupce těchto jazyků jmenuji *C++* nebo *Object Pascal*.

Při volbě jazyka však nesmíme zapomenout ani na *interpretační jazyky*, které taktéž mohou patřit do množiny objektově orientovaných. Tyto jazyky nevytvářejí strojový kód, nýbrž virtuální strojový kód, který není interpretován přímo procesorem, ale prostřednictvím tzv. *virtuálního stroje*. Virtuální stroj je speciální softwarová mezivrstva, jejímž primárním účelem je odstínit hardwarová specifika počítače. Vykonávání kódu probíhá na virtálním stroji, díky čemuž je dosaženo naprosté nezávisloti na konkrétní platformě. Výsledná aplikace běží pomaleji, než aplikace kompilovaná přímo do strojového kódu, ale její kód je zato (alespoň teoreticky) použitelný na různých typech procesorů a operačních systémů.

Jako mezikód se ve většině případů využívá tzv. *bytekód*, což je binární forma mezikódu s členěním po bytech. Bytekód je nezávislý na platformě, oproti přímé interpretaci ze zdrojových kódů je rychlejší a nevyžaduje překladač jako součást virtuálního stroje. Typickými představiteli interpretačních jazyků jsou *Java* nebo *C#*.

Příkladem je bytekód vytvářený programovacím jazykem *Java*. Se zajímavou nabídkou také přichází „.NET framework“ od firmy Microsoft, jehož výhodou je dostupnost kompilátorů pro více jazyků (*C++*, *C#*, *J#*, ...). Primární určení tohoto frameworku je bohužel pro operační systémy Windows. Tento nedostatek se snaží řešit opensource alternativa *Mono*, obsahující také překladač jazyka C#.

Vzhledem k rozvoji mobilních platforem a dalších elektronických zařízení, si dokáži živě představit, že v budoucnu nebude na vrátnici klasické PC, tak je je známe dnes. Již nyní jsou populární elektronické pokladny a jiná zařízení, která mohou mít různé hardwarové architektury. Pro zachování jakési univerzálnosti a přenositelnosti kódu, jsem se rozhodl jít cestou interpretačních jazyků. Nyní je třeba rozhodnout se mezi Javou a C#.

Jazyk *Java* nabízí jasné výhody:

- nezávislost na operačním systému (MS Windows i GNU Linux)
- profesionální vývojové prostředí je dostupné zdarma i pro komerční využití
- cena

To co je jsem zmiňoval jako výhody pro Javu, je pro .NET framework nevýhodou.

- přestože vznikají projekty jako je *Mono*, nelze hovořit o dostatečné nezávislosti na operačním systému. Mono je samozřejmě ve vývoji opožděné a už vůbec nelze mluvit o 100% kompatibilitě kódu. V tomto ohledu je *Java* jasnou jedničkou na trhu.
- pro vývoj aplikací v .NET frameworku je určeno *Visual studio*. Jedná se o velmi kvalitní vývojové prostředí, které nabízí vývoj v několika programovacích jazycích. Profesionální verze Visual Studia není zrovna levnou záležitostí. Pokud bychom chtěli zmenšit počáteční investice na vývoj, Microsoft nabízí ořezanou verzi Visual studia, která nese název *Visual Studio Express Edition*. Tato verze je sice ořezána o plnou funkcionalitu, nicméně je nabízena zdarma i pro komerční užití.

.NET framework však oproti Javě nabízí kompilátory pro více jazyků, což může znamenat pro firmy velkou výhodu. Představme si situaci, že má firma část programátorů, kteří jsou seznámeni s firemní problematikou a jejich znalosti jsou pro firmu velmi cenné. Tito programátoři ovládají jazyk *C++*. Nahrazení takového zaměstnance nepřipadá v úvahu a přeškolení na jiný programovací jazyk není krátkodobou záležitostí. .NET framework tyto problémy stírá. Část zaměstnanců může vytvářet moduly v *C++* a část v *C#*. Organizace a logické rozčlenění kódu už záleží na konkrétní firmě.

Mezi další výhody zmiňovaného .NET frameworku patří velmi silná podpora ze strany výrobce a user-friendly prostředí. Často se také setkávám s názorem: „Kdo jiný by měl mít lepší vývojové prostředí či programovací jazyk pro Microsoft Windows než samotný Microsoft?“

Cílem této kapitoly není popis vývojových prostředí do hloubky. Důležité bylo sdělit, jaké jsou možnosti a vybrat tu nejvhodnější. Po rozboru se jako nejvhodnější jevilo použití .NET frameworku v kombinaci s jazykem C#.

3.4 NUnit

NUnit[3] je *unit-testing* framework pro všechny .NET jazyky. Slovní spojení *unit testing* se používá jako pojem, protože v českém jazyce nemá ustálený překlad. *Unit testing* by se dal přeložit jako testování jednotek. Unit testing souvisí s vývojem programů. Koncoví uživatelé se s ním proto nesetkávají.

Jednotkový test je test pro určitou jednotku. V ideálním případě by měl být každý testovaný případ nezávislý na ostatních. Při testování se snažíme testovanou část izolovat od ostatních částí programu. Za tím účelem se někdy vytvářejí pomocné objekty, které simulují předpokládaný kontext, ve kterém testovaná část pracuje. Technika unit testing je jednou z klíčových součástí metodiky, která se nazývá *extrémní programování*.[29]

V praxi to znamená, vytvoření sady metod, které testují metody programu. Řekněme, že máme knihovnu *databaze.dll* s metodou *vyberAktualniZamestnance()*, která vrátí seznam aktuálních zaměstnanců. Pak vytvoříme knihovnu *databaze-tests.dll* s testovací metodou *testVyberAktualniZamestnance()*. Testovací metoda nejprve vloží seznam několika zaměstnanců, kteří musí být vyhodnoceni jako přítomní a několik zaměstnanců, kteří musí být vyhodnoceni, jako nepřítomní. Testovací metoda pak volá metodu *vyberAktualniZamestnance()* a **porovnává předpokládaný stav se skutečným**. Pokud je shoda předpokladů se skutečností, pak je vhodné, aby po sobě testovací metoda uklidila (smazala vložené testovací údaje).

3. Dostupné z `http://nunit.org`

Tímto způsobem se snažíme otestovat **všechny funkce** a zároveň **všechny stavy**, ve kterých se může funkce nacházet. Určitě mi dáte za pravdu, když řeknu, že ruční testování všech stavů jedné funkce je oproti této metodě naprosto neefektivní záležitost. V ideálním případě nejprve napíšeme testovací metody a pak teprve začneme vytvářet program. Často si tím ujasníme, co je vlastně náplní jednotlivých metod.

Časem v programovém kódu něco změníme, ale tato změna může nepředvídatelně ovlivnit další funkcie. Není nic jednoduššího, než spustit sadu testů, které ověří, že všechno funguje podle našich očekávání. Výsledkem je statistika, která řekne, kolik testů selhalo a z jakého důvodu.

NUnit samozřejmě není plnohodnotnou náhradou testera. Hodí se spíše na testování datových částí programu, grafické rozhraní tímto způsobem neotestujete. Další nevýhodou jsou počáteční investice - musíte naprogramovat testovací třídy.

Celkově je NUnit velmi kvalitní a užitečný nástroj. Z dlouhodobého hlediska vám kvalitní testování *zvýší efektivitu vývoje softwaru, ušetří čas a znatelně zredukuje počet chyb*.

3.5 ORM (Object Relational Mapping)

Objektově-relační mapování (ORM) řeší rozdílnost mezi objektovým a relačním paradigmatem a vytváří mezi nimi most. Tento most umožňuje v objektově-orientovaném programování vytvářet objektový kód, bez nutnosti psaní SQL příkazů. Potřebné SQL dotazy jsou automaticky generovány ORM systémem. Bez ORM řešení jsou vývojáři nuceni přistupovat ke stejným datům dvěma rozdílnými způsoby. Nejdříve je nutné vytvořit objektový programový model a pak tento model plnit daty z databáze, pomocí SQL dotazů. Díky ORM stačí vytvořit objektový programový model, do kterého budou data z databáze snadno namapována.

Vzhledem k existenci většího množství ORM nástrojů, stojíme před otázkou, který si vlastně vybrat? Pro některé bude první volbou LINQ to SQL. Toto řešení lze bohužel uplatnit pouze u jednoduchého mapování (1:1). Další nevýhodou LINQ to SQL je závislost na MS SQL, i když se nyní objevují snahy portovat tento nástroj i pro jiné databáze.

Proč používat ORM?

- Mezi největší výhody patří možnost pracovat s objekty a ne s SQL příkazy.
- Při použití vhodného ORM lze dosáhnout nezávislosti na typu databáze. ORM nástroj je vrstva nad databází - databázové dotazy generuje ORM. Záleží na použitém ORM, zda obsahuje logiku pro různé druhy databází. ORM je velmi vhodné pro produkty u nichž si klienti určují typ databáze.
- Query Language (QL), který je součástí ORM a slouží k dotazování na data v databázi, pracuje s objekty a ne s databázovými tabulkami.
- ORM sice představují určitou zátěž pro systém, ale u současných počítačových sestav to není nepřekonatelný problém.
- U většiny ORM je možné použít nativních SQL dotazů pro optimalizaci výkonu nad danou databází. ORM pak umožní snadno namapovat výsledek SQL dotazu zpět na objekty.
- ORM generují velmi dobré SQL dotazy a mohou být lepší než ručně vytvořený dotaz vývojáře, protože ORM často znají optimální varianty pro konkrétní databázi. Vývojáři tak nemusí být zrovna experty přes databáze.

Jaké jsou nevýhody?

- ORM nástroje jsou velmi komplexní a vyžadují dobré znalosti. Vznikají tak počáteční investice na pochopení a seznámení se s nástrojem.
- I když ORM generují velmi dobré SQL dotazy, nemusí být tyto dotazy vždy optimální a proto ORM stále vyžaduje znalost jazyka SQL.

- Je na zváženou, zda využít takto komplexní ORM nástroj u malého projektu. Režie na mapování a konfigurační nastavení může převýšit dobu, potřebnou na vytvořeních primitivních SQL dotazů.

3.6 NHibernate

NHibernate[4] je open source ORM nástroj licencovaný pod GNU Lesser General Public License. Projekt má široké zázemí a vznikl portací původního Hibernate, který je napsaný v Javě. **NHibernate je nezávislý na databázovém systému** a nevyžaduje žádné změny v objektovém modelu ani v databázi. Jednoduše si namapujeme databázové schéma bez nutnosti zásahu do struktury tabulek.

Mapování databázových tabulek na objektový model se provádí formou XML souborů, jehož nevýhodou je, že jejich psaní je zdlouhavé a do spuštění programu nemáme jistotu, že jsou tabulky namapovány správně. Vzniká nám tak další potenciální zdroj chyb. Jednou z možností, jak tyto chyby minimalizovat je použití různých generátorů, které automaticky vytvoří XMLka podle databáze. Tyto generátory však nevygenerují vše podle našich představ. Nakonec tak stejně sklouzneme k ručnímu zásahu do mapovacího XML souboru. Další alternativou, jak minimalizovat chyby při mapování, je použít projekt **Fluent NHibernate**, který umožňuje psát mapování přímo v jazyce C# a kontroluje tak správnost v době překladu.

Pro představu uvádím příklad jednoduchého mapování nejdříve pomocí XML souboru a posléze přímo v jazyce C# s využitím frameworku **Fluent NHibernate**. Budeme mapovat jednoduchou třídu *TaskMap*, která má 3 parametry: *Id* (primární klíč), *TaskName* a *Date*.

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
                   assembly="database"
                   namespace="database">
  <class name="TaskMap" table="TASKS">
    <id name="Id" column="Id">
      <generator class="assigned" />
    </id>
    <property name="TaskName" column="Name" length="35" />
    <property name="Date" column="Date" />
  </class>
</hibernate-mapping>
```

A nyní mapování stejné třídy za pomoci **Fluent NHibernate**:

```
public class TaskMap : ClassMap<Task>
{
    public TaskMap()
    {
        Id(x => x.Id);
        Map(x => x.TaskName);
        Map(x => x.Date);
    }
}
```

Při komunikaci našeho programu s databází nejprve vytvoříme sezení (session). Změny, které provedeme v rámci tohoto sezení nejsou ihned reflektovány do databáze. Očekává se, že změnu objektu potvrdíme ručně.

4. Dostupné z `http://www.nhibernate.org`

```
var config = new Configuration();
config.AddAssembly("NaseAssemblySMappingSoubory");
var factory = config.BuildSessionFactory();

using (ISession session = factory.OpenSession())
{
        TaskMap task = session.Get(1);  // načtení záznamu z databáze
        task.TaskName = "novejmeno";    // nastaví jméno záznamu z db

        session.Update(task);           // změna reflektována do db
        session.Flush();
}

factory.Close();
```

Lazy loading
NHibernate podporuje **lazy loading**. Lazy loading zajišťuje, načtení je jedné úrovně objektu z databáze. Pokud by měl náš objekt *TaskMap* atribut, který by byl taktéž typu *TaskMap*, nebude tento rekurzivní parametr načten. Abychom tento rekurzivní parametr načetli, museli bychom **lazy loading** potlačit, provést upravu mapování nebo explicitně požádat o načtení rekurzivního parametru. **Lazy loading se nedoporučuje vypínat.** Mohlo by dojít k situaci, že program bude neustále načítat celou databázi, místo toho, aby načetl data z tabulek, které potřebuje momentálně ke své práci.

Dotazování
Tak jako je SQL dotazovací jazyk u databází, tak jsou **HQL** a NHibernate **Criteria API** dotazovací jazyky nad objekty v NHibernate. Další možností, je využít vlastností jazyka LINQ.

3.7 LINQ

LINQ (anglicky Language Integrated Query) je integrovaný jazyk pro dotazování, který byl představen spolu s jazyky C# 3.0 a Visual Basic 9 spolu s .NET Frameworkem 3.5. LINQ přináší nový způsob pro dotazování nad libovolnými daty, usnadňuje jejich tvorbu, třídění, propojování i vyhledávání v nich.[25]

Existuje několik oficiálních implementací LINQ od Microsoftu, které by měly pokrýt běžné potřeby pro tvorbu aplikací, pracujících s daty.

- LINQ to Objects – Implementace LINQ pro standardní kolekce nacházející se v paměti
- LINQ to XML – Implementace LINQ pro práci s XML daty
- LINQ to DataSet – Implementace LINQ pro práci s ADO .NET datasety
- LINQ to SQL – Implementace LINQ pro Microsoft SQL Server 2000 a vyšší. Umožňuje dotazování nad databází MS SQL. MS SQL má vlastní dotazovací jazyk SQL, takže se ve skutečnosti všechny dotazy LINQu mapují na SQL příkazy. Skutečnou konverzi ale programátor nevidí. Výhodou použití LINQu je zejména objektový pohled na data.

Objevují se i další implementtace LINQu, například LINQ to Amazon, který slouží pro vyhledávání knih v tomto internetovém obchodě [5].

5. http://www.amazon.com/

Velká výhoda tohoto dotazovacího jazyka je, že odhalý chyby již v době kompilace. Drtivá většina chyb je tak odhalena, což se nám například při použití dotazovacího jazyka SQL nepodaří. Pomocí tohoto přístupu lze pracovat takřka s jakýmikoliv daty. Důsledkem toho je, že se **sjednocuje způsob práce s daty různých zdrojů či druhů.**

Přestože primárním přínosem dotazovacího jazyka LINQ efektivita, při dotazování nad daty, lze data také modifikovat. Například LINQ to SQL nám jednak umožňuje jednoduše a typově provádět dotazy nad databází SQL server a za druhé máme možnost modifikace těchto dat. Díky tomuto se z LINQ stává skvěle použitelný objektově-relační mapper.

Příklad použití LINQ nad kolekcí
Tento příklad ukazuje výběr všech řetězců, jejichž délka je menší než 5 znaků. Nakonec jsou řetězce setříděny podle abecedy.
Druhý řádek kódu ukazuje příklad použití dotazovacího jazyka LINQ. Řádek začíná klíčovým slovem var, což kompilátoru sdělí, že si má typ deklarované proměnné odvodit sám, z dalšího kontextu. Příkaz *from p in zakaznici* znamená, že všechna jména zákazníků budou postupně přiřazena do proměnné *p*. Klíčové slovo *where* pak omezí výběr zákazníků dle našich požadavků.

```
string[] zakaznici = {"Petr", "Pavel", "Mirek", "Jan", "Jiří"};
var jmena = from p in zakaznici
            where p.Length < 5
            orderby p
            select p;

foreach (string jmeno in jmena)
{
   Console.WriteLine(jmeno);
}

// výstupem budou jména v tomto pořadí: Jan, Jiří, Petr
```

Příklad použití Lambda výrazů
LINQ je možné použít i v jiné formě – pomocí rozšiřujících metod a **Lambda výrazů**. Zde je ukázka kódu, jehož výsledek je stejný, jako u předchozího příkladu.
Kód na druhém řádku *zakaznici.Where(p => p.Length < 5).OrderBy(p => p)* využívá *Lambda výrazů*. Za proměnnou *p* postupně dosadí jednotlivá jména z proměnné *zakaznici* a uplatní na nich podmínky.

```
string[] zakaznici = {"Petr", "Pavel", "Mirek", "Jan", "Jiří"};
var jmena = zakaznici.Where(p => p.Length < 5).OrderBy(p => p);

foreach (string jmeno in jmena)
{
   Console.WriteLine(jmeno);
}

// výstupem budou jména v tomto pořadí: Jan, Jiří, Petr
```

Deferred execution
Jednou z vlastnotí LINQ je, že poměrně velká část z množiny standard query operator metod není spouštěna ihned v době vytvoření LINQ dotazu. U předchozích příkladů není do proměnné *jmena* ihned dosazen vyhodnocený výsledek dotazu. K vyhodnocení dojde, až když je tomu potřeba. Tuto vlastnost oceníte zejména v implementaci LINQ to SQL. Zde nebude proveden dotaz do databáze, dokud ho nebude skutečně potřeba.

3.8 LINQ to NHibernate nebo DbLinq?

Dotazovací jazyk LINQ jsem shledal natolik užitečným, že jsem se rozhodl pro jeho použití. Je zde ale ještě jeden závažný problém. Použitím implementace **LINQ to SQL** od společnosti Microsoft by byl výsledný produkt svázán pouze s databází MS SQL. Jeden z mých cílů však hovoří o tom, že je třeba dosáhnout nezávislosti na typu databáze.

DbLinq
Projekt DbLinq[6] se snaží o implementaci LINQ to SQL hned pro několik databázových serverů: *Oracle*, *PostgreSQL*, *MySQL*, *Ingres*, *SQLite*, *Firebird*, *MS SQL*. DbLinq tedy z počátku vypadal na vhodného kandidáta. Při testování jsem však narazil na **několik vážných problémů, které vedly k zamítnutí tohoto projektu**:

1. jedná se o relativně mladý projekt, který nemá dostatečnou komunitní základnu a o budoucí podpoře projektu lze v současné chvíli pouze spekulovat.

2. nedostatečná dokumentace projektu.

3. v případě, že nebude implementace dotazovacího jazyka LINQ úplná, je nutné přikročit k dotazování pomocí jazyka SQL. Tím ztratíme možnost nezávislosti na typu databáze.

4. umožňuje pouze jednoduché mapování tabulek 1:1. Při složitější konstrukci databáze nebo větší programové abstrakci přicházíme o některé výhody. Například možnost definovat složitější integritní omezení v rámci mapování databáze.

Linq to NHibernate
Projekt Linq to NHibernate[7] (Linq2NHibernate) je nádstavba nad samotným *NHibernate*. Projekt byl sice v době psaní této práce ve stádiu alfa verze, nicméně už v současnosti se těší velkému zájmu.
Použití této nádstavby nad NHibernate řeší většinu problémů, které bránily využití *DbLinq*. NHibernate je široce užívaný *ORM* nástroj, má silné komunitní zázemí a je léty prověřený spoustou vývojářů. Umožňuje složité mapování a tím nabízí větší programovou abstrakci. Definice integritních omezení taktéž není překážkou. Linq2NHibernate bude nad tímto základem sloužit pouze k dotazování. Může se samozřejmě stát, že implementace Linq2NHibernate nebude úplná, nebo v Linq2NHibernate nebude možné vytvořit složitější dotaz. V tom případě se nestane nic horšího, než že bude nutné použít standardní dotazovací jazyk pro NHibernate - jazyk HQL. Tímto krokem však neztrácíme možnost nezávislosti na typu databáze.
Dotazovací jazyk LINQ je natolik zajímavé řešení, že jsem se rozhodl o jeho využití. Proto jsem se rozhodl použít projekt Linq2NHibernate i přesto, že je teprve ve stádiu alfa verze. Oproti projektu *DbLinq* však přináší nesporné výhody.

---

6. Dostupné z `http://code.google.com/p/dblinq2007`
7. `http://sourceforge.net/project/showfiles.php?group_id=216446&package_id=306405&release_id=654054to`

3.9 Log4net

Log4net[8] je nástroj licencovaný pod Apache License v. 2.0, který pomáhá se zápisem zpráv o činnosti programu do různých výstupních formátů. Log4net vznikl portací původního *log4j*, který je vyvíjen od roku 1996. Jedná se o prověřenou architekturu, která již byla portována do 12 různých jazyků. Cílem log4net je poskytnout nástroje, pro efektivní a snadné logování, pro účely debugování a auditu aplikací.

Log4net kategorizuje logování do těchto úrovní: DEBUG, INFO, WARN, ERROR a FATAL. Je optimalizovaný pro rychlé zpracování a lze měnit jeho nastavení za běhu laděné aplikace.

Níže uvádím příklad logování aplikace do textového souboru. Z příkladu je vidět několik úrovní logování, jaký byl sled událostí a mnoho dalšího. Formát logování u příkladu je následující:
datum [číslo vlákna] úroveň knihovna [doména uživatel@jméno-počítace [ip-adresa]] [jméno spuštěného okna] - samotná zpráva.

Je vidět, že se zaznamenává nemalé množství událostí. Každý vývojář si může zvolit, jaké informace zaznamenávat a úroveň jejich zaznamenávání (DEBUG, ERROR, ...). Kromě ukládání záznamu do textového souboru, nabízí log4net například záznam do: konzole, Windows, databáze, e-mailu, záznam přes síť a mnoho dalších. Veškeré nastavení pro logování je uloženo v textovém konfiguračním souboru, který je přikládán k vyvíjené aplikaci. Změnou textového souboru můžete změnit zaznamenávané informace, **bez rekompilace a to i za běhu laděné aplikace**.

```
2009-04-27 23:24:50,290 [1500] DEBUG gcs7900.Program [ASUS-GENTOO\
  nosek@ASUS-GENTOO[10.0.2.15]] [(null)] - INITIALIZATION.
2009-04-27 23:24:50,641 [1500] DEBUG language.Language [ASUS-GENTOO\
  nosek@ASUS-GENTOO[10.0.2.15]] [(null)] -
  C:\Projects\gcs7900\gcs7900\bin\Debug\lang\cs.xml
2009-04-27 23:24:51,933 [1500] INFO  NHibernate.Cfg.Environment
 [ASUS-GENTOO\nosek@ASUS-GENTOO[10.0.2.15]] [(null)] -
  NHibernate 2.0.1.4000 (2.0.1.4000)
2009-04-27 23:25:09,735 [1500] DEBUG gcs7900.Program [ASUS-GENTOO\
  nosek@ASUS-GENTOO[10.0.2.15]] [(null)] - END OF INITIALIZATION.
2009-04-27 23:25:10,066 [1500] DEBUG gcs7900.Program [ASUS-GENTOO\
  nosek@ASUS-GENTOO[10.0.2.15]] [(null)] - Program is starting.
```

3.10 MONO

Mono[9] je projekt vedený firmou Novell (dříve firmou Ximian). Jeho cílem je vytvořit sadu nástrojů kompatibilních s prostředím .NET, které splňují standardy ECMA (Ecma-334 a Ecma-335). K těmto nástrojům patří i překladač jazyka C# a Common Language Runtime. Mono může běžet na počítačích s operačními systémy Linux, FreeBSD, UNIX, Mac OS X, Solaris a Microsoft Windows. [26]

Cílem open source projektu Mono je vyvinout open source verzi vývojové platformy Microsoft .NET. Umožňuje tak vývoj .NET aplikací v Linuxu, potažmo usnadňuje portaci aplikací na jiný operační systém.

8. Dostupné z `http://logging.apache.org/log4net/index.html`
9. Dostupné z `http://mono-project.com`

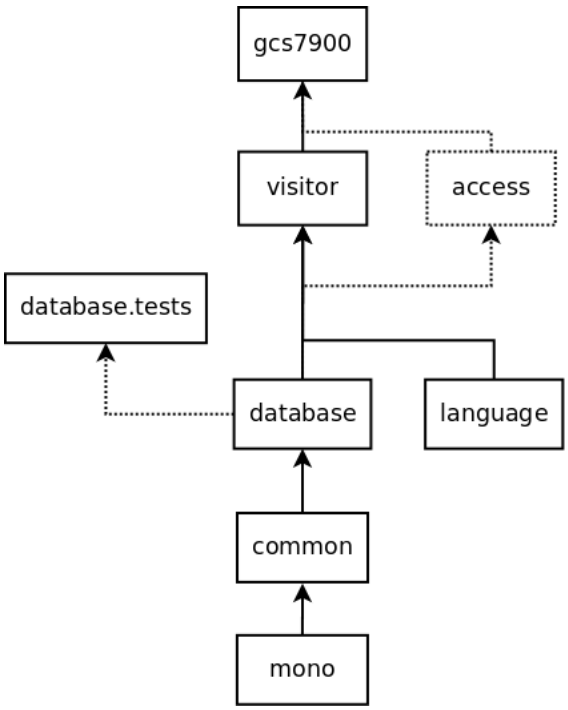
Kapitola 4

Implementace

Tato kapitola pojednává o konkrétní implementaci uživatelského rozhraní pro evidenci a řízení návštěv. Čtenář se zde dočte základní informace o jednotlivých knihovnách, které byly vyvinuty pro potřeby programu. Je zde rozebrána hierarchie těchto knihoven a jejich účel.

4.1 Hierarchie knihoven

Obrázek 4.1 popisuje hierarchii jednotlivých knihoven programu. Závislost mezi knihovnami je budována od spodu. Knihovna *mono* tak není závislá na žádné jiné programové knihovně. Stejně tak knihovna *language*, která leží v jiné úrovni hierarchie. Knihovna *common* je však závislá na knihovně *mono*. Na nejvyšší úrovni je knihovna *gcs7900*, která tvoří spustitelný program a využívá všechny programové knihovny.



Obrázek 4.1: Hierarchie knihoven

Na obrázku 4.1 jsou vidět dvě přerušované vazby. První vazba se týká knihovny *database.tests* a vyznačuje, že knihovna *database.tests* skutečně existuje, závislost je skutečná, nicméně knihovna *database.tests* není potřebná k práci programu. Tato knihovna obsahuje NUnitové testy, které slouží k otestování programu před uvolněním do produkční sféry.

Druhou přerušovanou vazbu tvoří knihovna *access*. Tato knihovna ve skutečnosti neexistuje, je vykreslena pouze pro snadnější pochopení filosofie programu.

Filosofie programu
Od počátku implementace je kladen důraz na modularitu programu. Celý program je dekomponován do několika knihoven a zároveň je **striktně oddělena databázová část a grafická nástavba** programu. Databázová část se nachází v knihovně *database* a grafická nástavba uživatelského rozhraní pro evidenci a řízení návštěv je obsažena v knihovně *visitor*.
Pokud by v budoucnu došlo k rozšíření systému *gcs7900*, například o modul přístupů, bude tento modul například naimplementován v knihovně *access*, jak je na obrázku zakresleno přerušovanými čarami. Knihovna *access* bude na stejné úrovni jako *visitor* a bude taktéž pracovat s funkcemi knihovny *database*. **Základní myšlenka spočívá v tom, že knihovny *visitor* a *access* budou obsahovat konkrétní grafické nádstavby, budou volat funkce ostatních knihoven na nižší úrovni a budou využívat stejné jádro programu.**
Současná dekompozice programu přináší ještě jednu velkou výhodu. Může se stát, že někdy v budoucnu vznikne požadavek na vytvoření uživatelského rozhraní ve formě *webové aplikace*. Nebude nic snažšího, než vytvořit nové grafické rozhraní, které bude optimalizované pro intranet. **Na toto optimalizované rozhraní se připojí již existující knihovny a databázové funkce.** Nebude tedy potřeba vytvářet webovou aplikaci od základních datových funkcí až po grafickou nástavbu. Z hlediska zdroje chyb bude u obou grafických nádstaveb stejné jádro – stejná datová knihovna.

4.2 Dokumentace

Dokumentace k vyvíjené aplikaci v jazyce C# se vkládá přímo do zdrojového kódu k třídám či metodám. Nejedná se o klasické řádkové nebo blokové komentáře, ale o XML komentáře. Komentářové značky jsou uvozeny sekvencí ///.

Důvody, proč psát komentářové značky jsou zřejmé:

- kód a dokumentace jsou na jednom místě, lze je snadno měnit obojí najednou
- vývojová prostředí zobrazují text dokumentačních komentářů v nápovědových bublinách
- usnadňujete práci sobě i ostatním vývojářům, kteří budou muset s kódem pracovat

Dokumentace napsaná ve zdrojovém kódu, má jednu velkou nevýhodu. Čtenář musí procházet všechny zdrojové kódy, aby našel konkrétní informaci. Dříve existoval nástroj *NDoc*, který veškerou dokumentaci vyextrahoval. V současné době jeho vývoj zamrznul. *NDoc* byl ale nahrazen novým dokumentátorem – Sandcastle[1].

Sandcastle je generátor dokumentace, který automaticky vyextrahuje dokumentační komentáře ze zdrojových souborů. Sandcastle je vlastně sada několika programů, konfiguračních souborů, komponent a XSLT transformátorů, které pracují dohromady tak, aby vyextrahovaly komentáře a vytvořily výslednou dokumentaci.

Sandcastle je konzolový nástroj. Pro snažší ovládání lze použít nádstavbu s grafickým uživatelským rozhraním – Sandcastle Help File Builder[2]. Dokumentace je generována do .chm souboru (HTML stránky jsou spojené do jednoho souboru, ve stromové struktuře s indexem) nebo do webové stránky.

1. Dostupné z `http://www.codeplex.com/Sandcastle`
2. Dostupné z `http://www.codeplex.com/SHFB`

Příklad komentářových značek ve zdrojovém kódu:

```
/// <summary>
/// Vrací řetězec se jménem počítače, na kterém byl program spuštěn.
/// </summary>
/// <remarks>
/// <para>
/// Pokud máte v síti počítač se jménem 'notebook', tato metoda
/// vrátí řetězec 'notebook'.
/// </para>
/// </remarks>
/// <returns>vrací jméno počítače</returns>
public string GetHostname()
{
    return Environment.MachineName.ToString();
}
```

4.3 Knihovna mono

V případě, že by Mono[3] bylo plně kompatibilní s .NET frameworkem, byla by existence této knihovny nadbytečná. Vyskytují se zde bohužel rozdíly[4], které se knihovna *mono* snaží sjednotit. Zejména se jedná o rozdíly mezi grafickými komponentami. Například pozicování komponenty *TableLayoutPanel* chápe Mono jinak, než .NET framework.

```
public static bool IsRunningOnMono()
{
    return Type.GetType("Mono.Runtime") != null;
}
```

Program na základě této funkce vyhodnotí, na které platformě běží – jestli Mono nebo .NET Framework. Pokud funkce vrátí logickou hodnotu TRUE, program běží na platformě Mono a bude aktivováno přizpůsobení příslušných komponent. V opačném případě se nic nestane, protože .NET Framework je považován za implicitní, chceme-li vzorovou, platformu.

Toto řešení má své nevýhody. Vývojář musí mít na paměti, že při návrhu grafického rozhraní nemůže používat komponentu *TableLayoutPanel*, ale musí používat komponentu *MonoTableLayoutPanel* z programové knihovny *mono*. V případě, že nová verze projektu Mono bude chápat pozicování jinak, než současná verze projektu Mono, bude zapotřebí odladʼovat program pro každou verzi projektu Mono zvlášť.

Na druhou stranu, pokud chceme mít program skutečně multiplatformní, vyplatí se věnovat úsilí pro vytvoření kvalitní knihovny, která rozdíly v chování frameworků sjednotí. **Úpravou jedné knihovny dosáhneme změn, které se projeví v celém programu, bez nutnosti dalšího ladění na vyšších vrstvách hierarchie knihoven.**

Nejlepším řešením pro multiplatformní software by bylo použít *Gtk# for .NET*, což je sada grafických komponent alternativní k *Windows Forms*. *Windows Forms* jsou součástí .NET Framework. *Gtk# for .NET* je narozdíl od *Windows Forms* multiplatformní. Cílem ale bylo otestování současného stavu projektu Mono a jeho vlastní implementace *Windows Forms*.

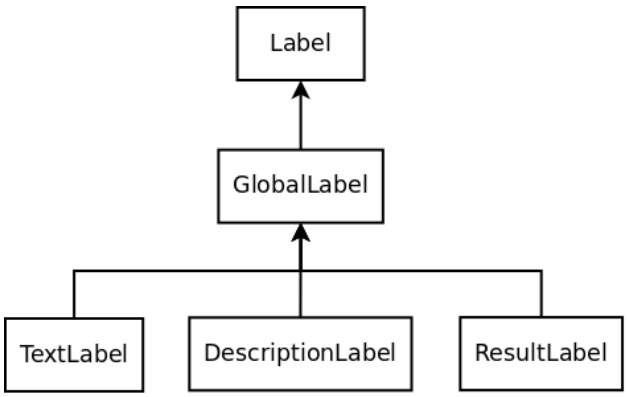
3. Dostupné z `http://mono-project.com`
4. Nalezené rozdíly byly odladěny a testovány pro Mono v. 2.4.

Aktuální stav programu
V současné chvíli je **grafické rozhraní** programu plně odladěno pro .NET Framework a částečně pro projekt Mono. Problémy nastaly ve chvíli, kdy jsem připojil grafické rozhraní k datové knihovně. Domnívám se, že zdrojem problému je komponenta *BindingSource*, která nemusí být v projektu *Mono* plně implementována. Pokud program běží pod projektem Mono, je funkční pouze hlavní okno a okno s historií návštěv. Tento problém jsem doposud neodstranil, jednak z důvodu možné časové náročnosti a za druhé nepatří optimalizace programu pro Mono mezi hlavní cíle této práce. Položil jsem však základní kameny pro možnou multiplatformnost.

4.4 Knihovna common

Knihovna obsahuje obecné funkce, které mohou být využity v rámci celého programu. Kromě obecných funkcí obsahuje i globální proměnné. Těchto globálních proměnných se využívá například při inicializaci programu. Při spuštění programu se například zjistí IP adresa počítače, cesta, ze které je program spouštěn či proměnné z databáze. Všechny tyto informace jsou uloženy v globálních proměnných. Zároveň jsou zde umístěny různé programové konstanty.

Grafické komponenty
Důležitou součástí knihovny jsou odvozené grafické komponenty. Program nevyužívá žádných základních grafických komponent. Použity jsou pouze grafické komponenty, které jsou odvozeny od základních komponent ze sady *Windows Forrms*. Typickým příkladem je komponenta *Label*. Hierarchii dědění znázorňuje obrázek 4.2. Komponenta *Label* patří do základní sady *Windows Forms*. Ostatní komponenty jsou v knihovně *common*.



Obrázek 4.2: Hierarchie grafické komponenty Label

Hlavním důvodem, který mě vedl k tomuto členění, je **jednoduchost hromadných úprav a jednotnost vzhledu**. Představme si, že se uživatelé programu rozhodnou, že chtějí všechny textové popisky (komponenta *Label*) červenou barvou. V běžné situaci budete muset projít celý kód programu a všude tam, kde se vyskytuje komponenta *Label*, musíte přidat řádek o barvě písma. Při členění stejném, jako na obrázku 4.2, stačí v konstruktoru třídy *GlobalLabel* přidat řádek s informací o barvě písma. Barva písma se pak projeví v celém programu i dalších podtřídách.
Podtřídy třídy *GlobalLabel* mají různá specifika, pro určité skupiny popisků. *TextLabel* se například používá u formulářů, kdy se za popisek automaticky vloží dvojtečka. Pro lepší pochopení uvádím programovou ukázku dvou zmiňovaných tříd *GlobalLabel* a *TextLabel*.

```
public class GlobalLabel : Label
{
    public GlobalLabel()
    {
        AutoSize = true;
        Anchor = AnchorStyles.None;
    }
}

public class TextLabel : GlobalLabel
{
    public override string Text
    {
      get  { return Conversion.Default.FirstToUpper(base.Text) + ":"; }
      set  { base.Text = value;}
    }
}
```

4.5 Knihovna language

Program je multilingvální a za jazykové mutace odpovídá právě knihovna language. **Jazyková mutace pro český jazyk je nadefinována v jednom XML souboru.** Jeho struktura vypadá takto:

```
<?xml version="1.0" encoding="utf-8" ?>
<language>
  <visitor>
    <MainWindowCaption>Recepce - návštěvy</MainWindowCaption>
    <TSNewVisit>Nová návštěva</TSNewVisit>
  </visitor>
</language>
```

Kořenový tag <*language*> určuje, že se jedná o soubor s jazykovou mutací. Druhá úroveň zanoření, tag <*visitor*> určuje, kterému modulu (sekci) bude příslušet. Když si připomeneme hierarchii knihoven z obrázku 4.1, vidíme, že na druhé úrovni zanoření by mohl být tag <*access*>.

Třetí úroveň zanoření už obsahuje klíčová slova a textový řetězec, který náleží příslušné jazykové mutaci.

Druhou úroveň zanoření jsem vytvořil z důvodu přehlednosti při programování. Předpokládal jsem, že v budoucnu bude existovat více uživatelských rozhraní (visitor, access) a mohlo by tak docházet k záměně klíčových slov ze třetí úrovně.

Jako příklad uvádím část programového kódu, kdy je volán popisek klíčového tagu *MainWindowCaption*, ze sekce *visitor*.

```
// inicializace - při startu programu
Language.InitializeSingeltonDefault("C:\Program\cs.xml");

// vrátí textový řetězec: Recepce - návštěvy
Language.Default["visitor","MainWindowCaption"];
```

Program sice nemá žádné aktivní tlačítko, které by přepínalo mezi různými jazykovými mutacemi, nicméně díky této knihovně bude tvorba takového tlačítka velmi snadná. Každá jazyková mutace bude v samostatném XML souboru.

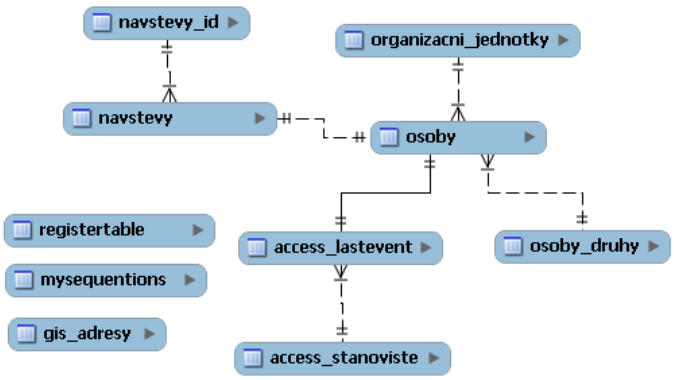
Pro vznik jazykové knihovny jsem měl ještě jeden důvod. Některé textové popisky se vyskytují na více místech v programu. Změnou jednoho popisku v XML souboru dosáhnu změny na všech místech, kde se řetězec vyskytoval. K těmto úpravám dochází zejména v situacích, kdy vývojář programu vytváří popisky, které jsou nesrozumitelné pro běžného uživatele. Zároveň vzniká snaha o **používání stejné terminologie v rámci celého programu**. Nemělo by docházet k situacím, kdy bude v hlavním okně programu kolonka recepční, v dalším okně kolonka zaměstnanec, přičemž recepční i zaměstnanec bude v terminologii programu jedna a tatáž osoba. Správně by mělo být v obou kolonkách programu napsáno recepční nebo zaměstnanec. **Jednotná terminologie vede k přehlednosti programu.**

4.6 Knihovna database

Tato knihovna je základem pro komunikaci s databází. Díky tomu, že využívá *NHibernate*, je nezávislá na databázovém serveru. Vývoj včetně testování proběhl na databázi Firebird[5]. Kromě toho, že je zde umístěno mapování na databázové tabulky, je součástí knihovny implementace rozhraní dle obrázku 6.2. Zmiňované rozhraní posléze využívá i grafická nástavba.

Program není s databází spojen permanentně, udržuje spojení pouze po dobu nezbytně nutnou pro vykonání SQL dotazů. Je to vlastně stejný přístup, jako při načítání webové stránky. Po načtení stránky se klient od serveru odpojí. Tento postup však přináší jeden problém - **lazy loading**. Při vykonání dotazu se z databáze načtou namapované objekty pouze do první úrovně. Tento problém byl podrobně rozebrán v předcházející kapitole při popisu NHibernate. Řešením je třída **LazyInitializer**, která inicializuje načítaný objekt do **libovolně požadované úrovně**. Při používání třídy je třeba dbát opatrnosti. Pokud je úroveň příliš vysoká, může dojít ke značnému zpomalení programu, **protože program bude z databáze načítat velký objem dat**.

Databáze
Tabulky, které jsou mapovány z databázového serveru, **neobsahují žádné integritní omezení**. Integritní omezení byla definována pouze slovně či písemně a z historických důvodů nebylo možné na této situaci nic změnit. Integritní omezení jsem tedy nadefinoval programově tak, aby odpovídala obsahu databáze. Na obrázku 4.3 jsou vidět všechny tabulky, které byly namapovány, včetně jejich závislostí v programu. Podrobné schéma včetně primárních klíčů a sloupců je umístěno v příloze, viz obrázek 6.1.



Obrázek 4.3: Schéma tabulek v databázi

Z obrázku je vidět, že některé tabulky jsou nezávislé. Tyto tabulky obsahují buď pevně stanovenou konfiguraci nebo tabulky pro pomocné operace. Názvy tabulek a sloupců působí v mnoha

5. Dostupné z `http://www.firebirdsql.org`

případech nelogicky a neznalý programátor nemá šanci do tohoto schématu jednoduše proniknout. Jak už jsem řekl dříve, na databázovém schématu není možné nic měnit. Programové schéma jsem upravil tak, aby bylo mnohem logičtější. Z programových tříd se čtenář velmi snadno dovtípí, jaké informace databázové tabulky obsahují.

databázová tabulka	**programová třída**	**popis**
navstevy_id	Company	Obsahuje nějaký identifikační údaj o návštěvě. Ve většině případů se jedná o jméno firmy. Záleží však na obsluze, jaké informace vloží.
navstevy	Visit	Jeden řádek tabulky nese základní informace o příchozí návštěvě. Například příchod, odchod atd.
osoby	Person	Jeden řádek tabulky obsahuje základní informace o jedné osobě. Například jméno, příjmení atd. Návštěva je speciální druh osoby. Stejně tak karta je pouze virtuální a z hlediska systému se jedná o speciální druh osoby.
osoby_druhy	PersonType	Určuje druh osoby. Například zaměstnanec, návštěva, virtuální návštěvnická karta.
organizacni_jednotky	Resort	Obsahuje oddělení, do kterých může být osoba přiřazena.
access_lastevent	IdEvent	Jeden řádek v tabulce odpovídá jednomu řádku v tabulce osoby. V tabulce access_lastevent je uložena informace o poslední operaci, kterou příslušná osoba provedla.
access_stanoviste	Station	Obsahuje názvy jednotlivých stanovišť. Například vrátnice, ředitelna atd.

Tabulka 4.1: Mapování jednotlivých tabulek

Rozhraní
Schémata rozhraní a jejich metod, která jsou v knihovně implementována, jsou zobrazena v příloze na obrázku 6.2. Popisovat jednotlivé metody rozhraní nemá smysl, protože již z názvů metod lze většinou pochopit jejich účel. Za povšimnutí stojí především skutečnost, že většina rozhraní implementuje rozhraní *IDao*. *IDao* obsahuje metody pro práci s databází – jako je uložení, úprava či mazání záznamu.

4.7 Knihovna database.tests

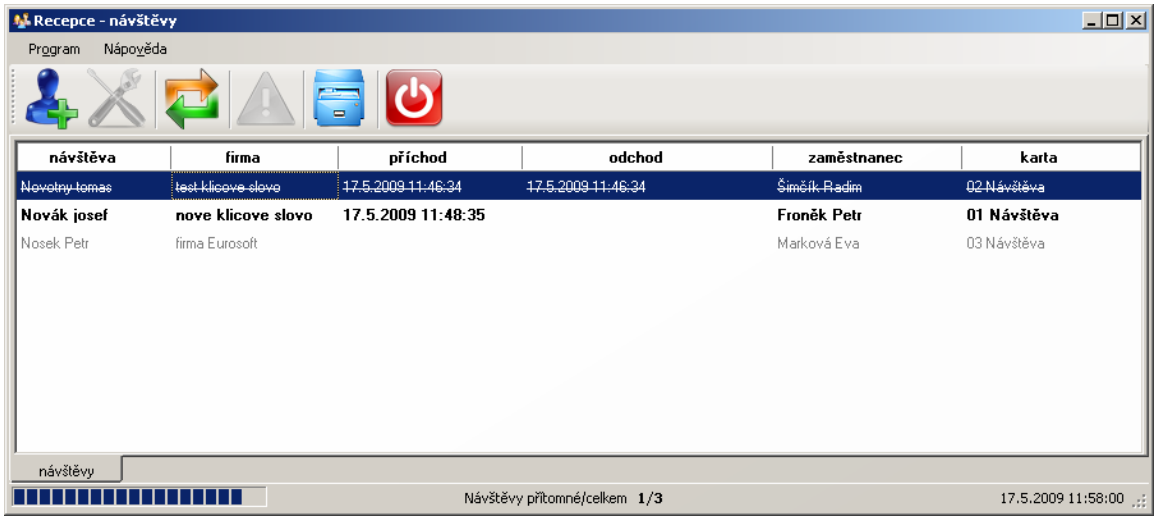
Tato knihovna je pouze pro testovací účely – k práci programu není nezbytná. Obsahuje testovací funkce pro implementované rozhraní v knihovně database. Celkem testuje 90% metod. Vždy, když jsem upravil některou metodu z knihovny database, spustil jsem všechny testy, abych se ujistil, že změna jedné metody nevyvolala neočekávané změny v metodách jiných. Testy jsem prováděl pomocí programu *NUnit.exe*, který je součástí projektu NUnit. Podrobnější informace o projektu NUnit jsou v předchozí kapitole.

4.8 Knihovna visitor

V této knihovně je implementováno celé grafické rozhraní pro evidenci a řízení návštěv. Po spuštění programu se načte hlavní okno podle obrázku 4.4. Největší část okna tvoří tabulka s aktuálními návštěvami. Na obrázku jsou vidět tři evidované návštěvy. První návštěva byla aktuálně ukončena. Řádek po vypršení časové konstanty zmizí a informace o této návštěvě půjdou dohledat v historii. Druhý řádek informuje o tom, že panu Novákovi byla přidělena karta a pan Novák již vstoupil do objektu. Třetí řádek informuje, že panu Noskovi byla přidělena karta, ale on stále ještě nevstoupil do objektu. Ve chvíli, kdy návštěva projde s kartou kolem vstupní čtečky, zaznačí se datum příchodu. Návštěvy jsou řazeny ve stejném pořadí, v jakém byly zapsány.

Čtečky karet jsou nastaveny tak, že zapisují data do databáze nezávisle na programu pro evidenci návštěv. Změny v databázových tabulkách však musí být reflektovány do grafického rozhraní. V ideálním případě databázový server oznámí evidenci návštěv, že došlo ke změně tabulek a evidence návštěv si vyžádá konkrétní změny. Tato problematika by si však zasloužila hlubší studium i z toho důvodu, že bych chtěl zachovat nezávislost na typu databázového serveru. Pro otestování programu, jsem do levého dolního rohu umístil progress bar, který zobrazuje odpočítávání času do dalšího obnovení dat v tabulce. Ikona dvou cyklících šipek slouží pro případy, že recepční potřebuje okamžitě načíst aktuální data.

Pokud návštěva opouští objekt, může se stát, že si nezaznačí odchod – například v případě ztráty karty. Z tohoto důvodu je v programu nezbytná funkce pro ruční ukončení návštěvy (trojúhelník s vykřičníkem). Na obrázku 4.4 je vidět, že označená návštěva je již ukončena – tlačítko pro ukončení je tedy neaktivní.



Obrázek 4.4: Hlavní okno uživatelského rozhraní

Mezi nejsložitější implementační část, patří vyhodnocování přítomnosti jednotlivých návštěv. Je zde totiž několik faktorů, na které je třeba brát zřetel:

1. Návštěvnické karty jsou předem nadefinovány a dochází k jejich recyklaci. Stejnou kartu, kterou má dnes pan Novák, může mít zítra pan Marek.

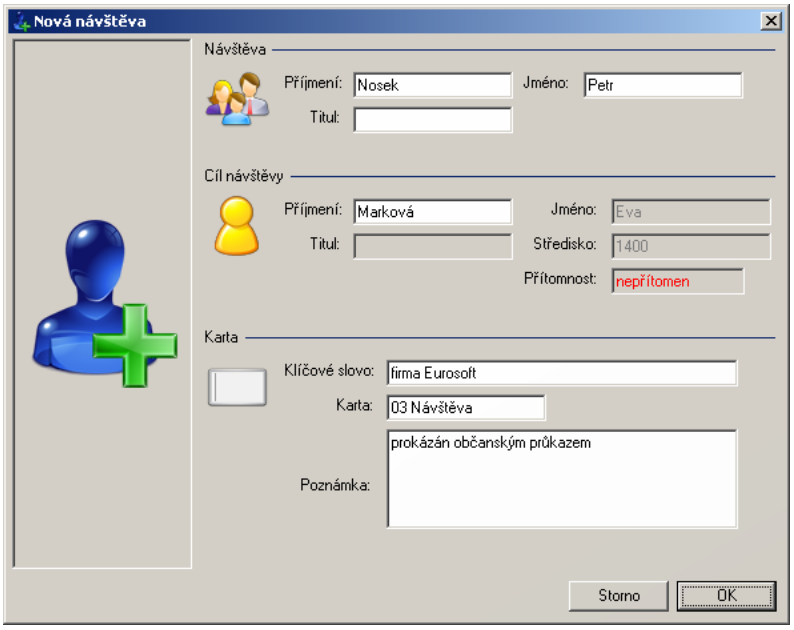
2. Při vyhodnocování času (například vstup) jsem musel brát v úvahu fakt, že čas na pracovních stanicích není synchronizován s časem databázového serveru a časem u čteček karet. Když program vyhodnocuje aktuální návštěvy podle času na pracovních stanicích, musí počítat s určitou časovou rezervou.

3. Některé karty mohou být nastaveny tak, že nesmí dojít k automatickému ukončení návštěvy. Obsluha vyžaduje, aby karty byly blokovány ručně.

4. Jenom některé čtečky karet jsou odchozí. Například na vrátnici je odchozí čtečka, před ředitelnou ne.

Přidání nové návštěvy
Recepční obsluhuje vždy nakonfigurovaný systém s předchystanými kartami. Nové návštěvě přidělí návštěvnickou kartu a vyplní základní údaje podle obrázku 4.5. Na obrázku je vidět, že se automaticky vypisuje přítomnost navštěvovaného zaměstnance. Ve formuláři pro přidávání návštěv je implementovaný našeptávač.
Když obsluha vyplňuje například příjmení navštěvované osoby, zobrazují se po stisku každé klávesy všechny možné volby. Když například bude v databázi zaměstnanců Petr Nosek, Jaroslav Novák a recepční do pole příjmení zadá „No“ - nabídnou se obě jména. Pokud však bude zadáno „Nos“, zobrazí se pouze možnost Petr Nosek. Pro zobrazení našeptávače stačí zadat pouze část jména nebo příjmení.

Další funkce
Program kromě vkládání nové návštěvy umožňuje úpravu již existujícího záznamu. Podmínkou upravení již vloženého záznamu je, že návštěvník ještě nepoužil kartu pro vstup do objektu. Pokud by tato podmínka nebyla splněna, mohla by recepční úmyslně nebo omylem měnit všechny záznamy. Program implementuje také funkcionalitu historie návštěv, kde je možné zobrazovat záznamy podle různých filtrů. U vyhledané náštěvy je pak možné zobrazit detail, jak ukazuje obrázek 4.6.



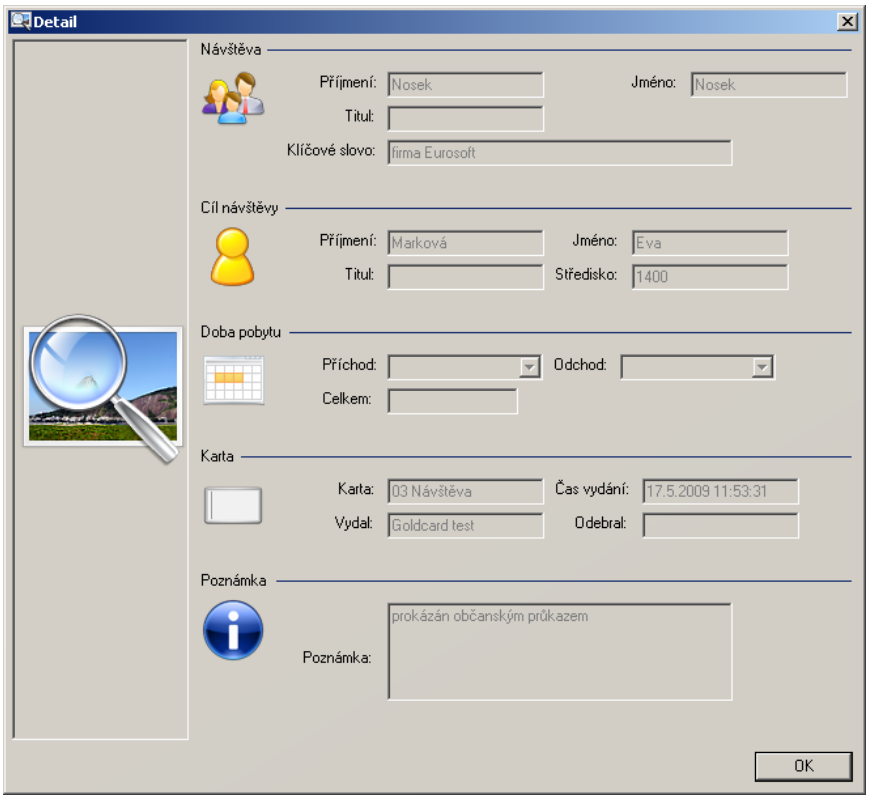
Obrázek 4.5: Přidání nové návštěvy

4.9 Gcs7900

Gcs7900 není knihovna, ale spustitelný soubor, který na začátku postupně inicializuje všechny knihovny a načte úvodní obrazovku modulu návštěv, dle obrázku 4.4. K tomuto spustitelnému souboru jsou přiloženy tři konfigurační soubory:

1. **gcs7900.exe.config** – Obsahuje informace o připojení k databázi (implicitně Firebird, ale může být změněno na libovolnou databázi, kterou podporuje projekt NHibernate[6]), cestu k jazykové mutaci a programové konstanty.
2. **gcs7900.exe.log4net** – Obsahuje informace zda mají být zaznamenávány ladící informace, jaká úroveň ladění má být zvolena a cíl, kam se budou soubory ukládat. Konkrétní příklady možných konfigurací jsou na stránkách projektu log4net[7].
3. **cs.xml** – Soubor s českou jazykovou mutací.

K projektu jsou samozřejmě přiloženy všechny licenční ujednání použitých knihoven.



Obrázek 4.6: Detail návštěvy

Ochrana osobních údajů

Z právního hlediska je při používání programu třeba brát zřetel na ochranu osobních údajů[8]. V zákonu se přímo uvádí, že firma „může shromažďovat osobní údaje odpovídající pouze stanovenému účelu a v rozsahu nezbytném pro naplnění stanoveného účelu“[22]. Z definice vyplývá, že dobu uchování historie návštěv, si musí stanovit každá firma sama, na základě jejího stanoveného účelu. V souboru s konfigurací je konstanta, která určuje, po jaké době se mají staré záznamy o návštěvách smazat. Implicitně je přednastaveno 24 hodin.

6. Dostupné z `https://www.hibernate.org/361.html`
7. Dostupné z `http://logging.apache.org/log4net/release/config-examples.html`
8. Dostupné z `http://www.uoou.cz`

Kapitola 5

Závěr

V této práci jsem si vytyčil pět důležitých cílů, které vyplynuly ze zadání a postupné analýzy projektu. Nejprve jsem provedl analýzu dostupného softwaru pro evidenci a řízení návštěv a konkrétního systému GCS7800 podle prvního bodu zadání. O této analýze podrobně pojednává druhá kapitola. Na druhou kapitolu volně navazuje kapitola třetí, kde popisuji všeobecné požadavky na systém evidence návštěv, diskutuji nad možnými implementačními platformami a stanovuji pět důležitých cílů, jimiž se bude projekt dále ubírat. Třetí kapitola taktéž popisuje vhodné implementační nástroje, jejich výhody či nevýhody a stanovuje řešení problematiky, což odpovídá druhému a třetímu bodu zadání. Samotnou implementaci a problémy, se kterými jsem se setkal, popisuji v kapitole číslo čtyři.

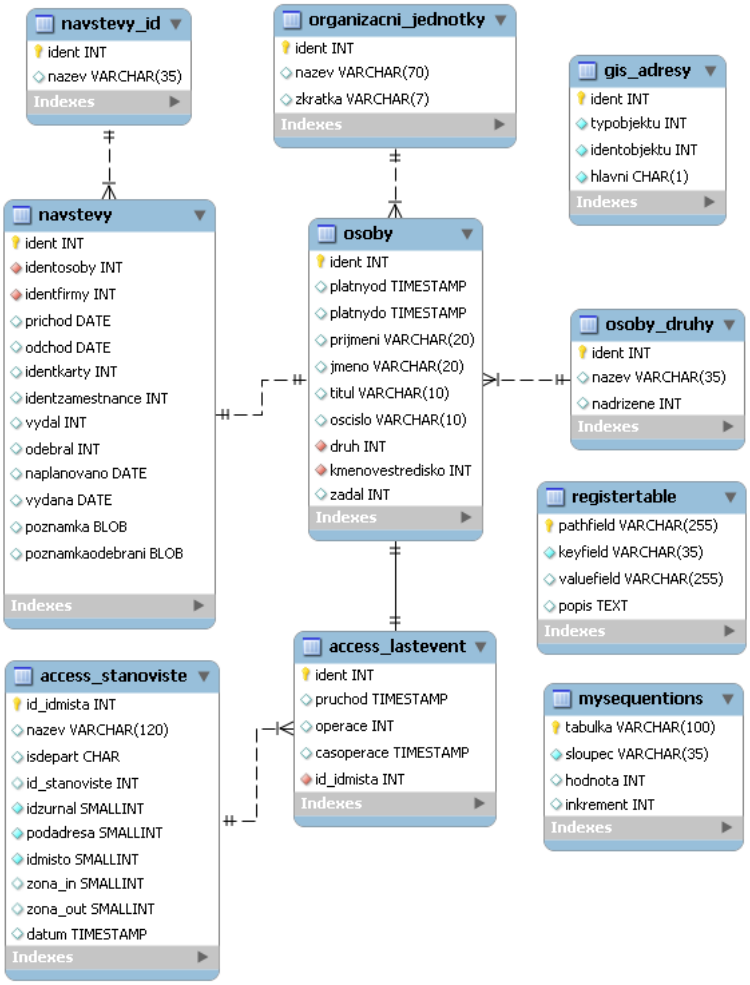
Jsem přesvědčen, že díky modularitě projektu a promyšlení jednotlivých bloků programu, jsou položeny základy pro další programový rozvoj. Programový kód je dostatečně zdokumentovaný a software je nezávislý na databázovém serveru. Pro uživatele jsem vytvořil intuitivní grafické rozhraní, které je vhodně doplněno napovídajícími ikonami. Taktéž jsem v grafickém rozhraní několikrát použil podobná formulářová okna pro podobné operace. Uživateli tak stačí osvojit si jedno z těchto formulářových oken a v dalším už nachází podobnost. Velké množství chyb se mi podařilo eliminovat už v rámci vývoje a to zejména při komunikaci s databázovým serverem. Psaní testovacích tříd pro databázovou část programu sice zabralo určité režijní náklady, nicméně tyto náklady se posléze mnohonásobně vrátily v podobě uspořeného času, který bych v opačném případě strávil hledáním jendotlivých chyb a jejich odstraňováním. Použitím verzovacího systému jsem dal projektu jasnou historii vývoje a přehlednost v jednotlivých verzích. V neposlední řadě je třeba zmínit logovací systém, který důsledně zaznamenává důležité údálosti v programu.

Program čítá přibližně *10 000* řádků programového kódu, včetně komentářů. Velikost programu včetně všech knihoven je *3 626* KB.

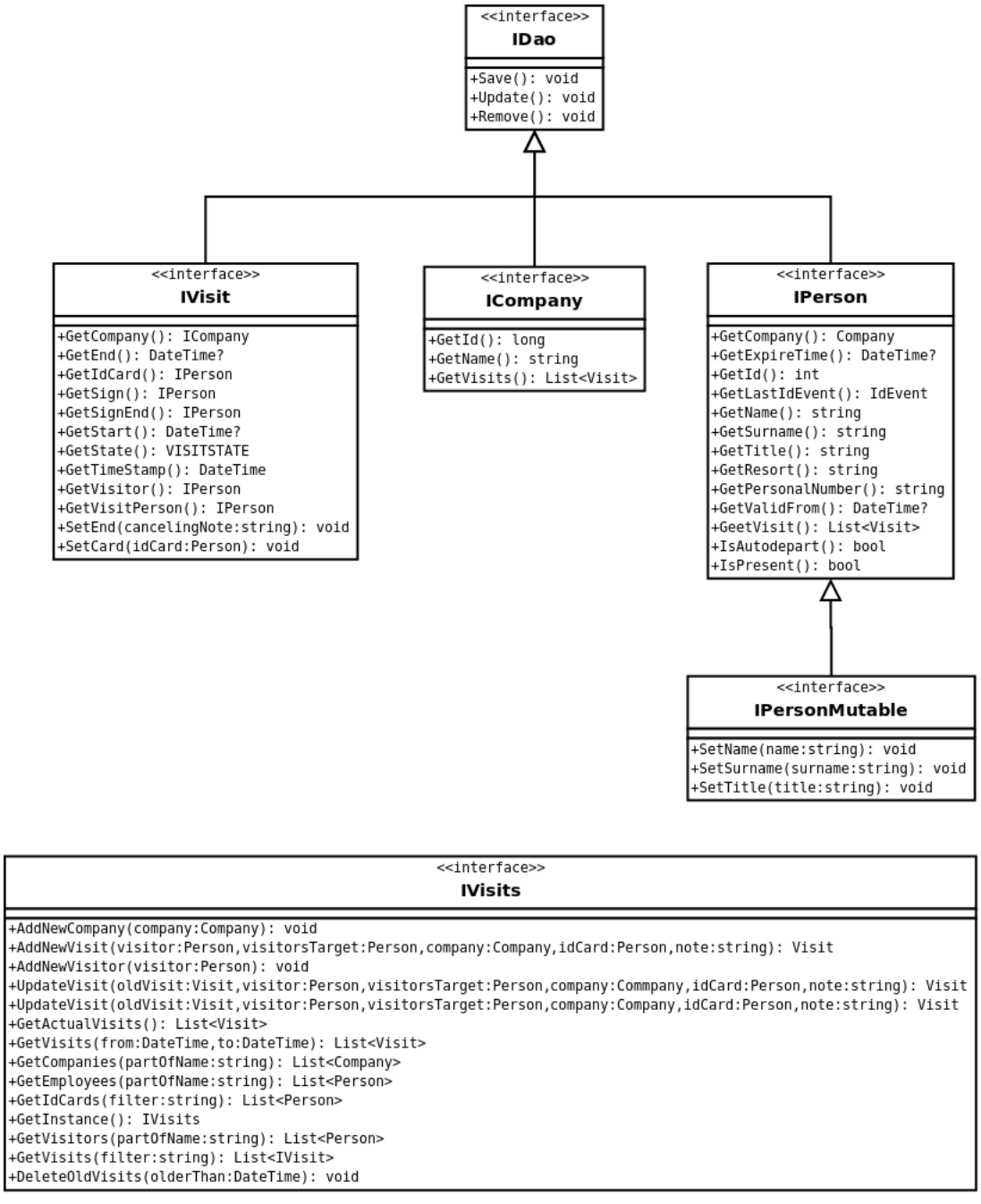
Všech stanovených cílů se mi podařilo úspěšně dosáhnout. Největším problémem byla počáteční časová investice do promyšlení struktury, studia a vytváření základních knihoven. Tyto investice se mi vracely až v polovině projektu a to ve formě rychlého postupu prací a snížené chybovosti aplikace. Projekt má všechny dispozice k tomu, aby byl do budoucna dále rozvíjen. Do budoucna by se dala zlepšit podpora pro jiné operační systémy, šifrovat heslo pro připojení k databázovému serveru, vytvořit další moduly – například přehledy o průchodech zaměstnanců na jednotlivých stanovištích, analyzovat výsledky chybovosti v produkčním prostředí, vytvořit autentizaci a zobrazovat přítomnost současných zaměstnanců či umožnit návštěvy plánovat dopředu.

Kapitola 6

Přílohy



Obrázek 6.1: Databázové schéma



Obrázek 6.2: Datový model rozhraní

Literatura

[1] *ANeT-Advanced Network Technology - vrátnice* [online]. ANeT, 2008. [cit. 25. 10. 2008]. Dostupné z: http://www.anet.info/systemy/aplikace-vratnice/.

[2] *Trade FIDES, a.s. - TORINET - modul MONITOR* [online]. Trade FIDES, 15.2.2007. [cit. 26. 10. 2008]. Dostupné z: http://www.fides.cz/cz/torinet-monitor.

[3] *Docházkové systémy* [online]. GOLDCARD, 2006. [cit. 3. 11. 2008]. Dostupné z: http://www.goldcard.cz/?page=systemy.

[4] *RON Software - docházka, mzdy, jídelna, majetek* [online]. RON SOFTWARE, 2003-2007. [cit. 31. 10. 2008]. Dostupné z: http://www.ron.cz/_cze/cze_dochazka_04rozsireni_03visitor.php.

[5] *Kniha návštěv v systému Docházka M.S.O.* [online]. Z-WARE, 2006-2008. [cit. 31. 10. 2008]. Dostupné z: http://www.z-ware.cz/?5-dochazka-mso-kniha-navstev.

[6] BAREŠ, I. J. *ORM - základní informace* [online]. 2009. [cit. 21. 4. 2009]. Dostupné z: http://jan.baresovi.cz/dr/cs/ORM-zakladni-informace.

[7] BAYER, J. *C# 2005 velká kniha řešení.* Vyd. 1. Brno : Computer Press, 2007. 813. s. Přel. z: Das C# Premium Codebook. ISBN 978-80-251-1620-3.

[8] CZILLA. *Jak správně oznamovat chyby* [online]. CZilla.cz, 2008. [cit. 20. 7. 2008]. Dostupné z: http://www.czilla.cz/czilla/jak-pomoci/jak-oznamovat-chyby.html.

[9] GRÉK, J. *Hibernate a objektově-relační paradigma* [online]. VSE.cz, 2005. [cit. 21. 2. 2009]. Dostupné z: http://nb.vse.cz/~zelenyj/it442/eseje/xgrej08/hiberorm.htm#uvod.

[10] HAUNER, A. *Bug tracking pro vývojáře* [online]. SoftEU.cz, 2008. [cit. 20. 11. 2008]. Dostupné z: http://blog.softeu.cz/prednasky/2008/prednaska-bug-tracking-pro-vyvojare/.

[11] J., K. *Evidence návštěv VISIT* [online]. COMINFO, 11.7.2006. [cit. 26. 10. 2008]. Dostupné z: http://www.cominfo.cz/PDF/visit_CZ.pdf.

[12] KOUBA, T. J. *Základní práce s NUnit 2.1.4* [online]. Tomas.NET, 2004. [cit. 10. 9. 2008]. Dostupné z: http://tomas-net.blogspot.com/2004/07/zkladn-prce-s-nunit-214.html.

[13] MALÝ, J. *Sandcastle - generování dokumentace z XML komentářů (nástupce NDoc)* [online]. 2007. [cit. 27. 2. 2009]. Dostupné z: http://blog.jakubmaly.cz/programovani/csharp-aspnet/sandcastle-generovani-dokumentace-z-xml-komentaru-nastupce-ndoc.aspx.

[14] MICROSOFT. *101 LINQ Samples* [online]. Microsoft Corporation, 2009. [cit. 25. 2. 2009]. Dostupné z: http://msdn.microsoft.com/en-us/vcsharp/aa336746.aspx.

[15] MOUDRÝ, J. *Welcome to DbLinq* [online]. Google Code, 2009. [cit. 27. 2. 2009]. Dostupné z: http://code.google.com/p/dblinq2007.

[16] NOVELL. *Projekt Mono dále posouvá možnosti vývoje na platformě .NET na Linuxu* [online]. Novell, Inc., 2008. [cit. 27. 2. 2009]. Dostupné z: http://www.novell.cz/cs/aktuality/projekt-mono-dale-posouva-moznosti-vyvoje-na-platforme-net-na-linuxu.html.

[17] PETZOLD, C. *Windows Forms v jazyce C#.* Brno : Computer Press, 2006. 356. s. Přel. z: Programming Microsoft Windows Forms. ISBN 80-251-1058-3.

[18] PUŠ, P. *Co je LINQ?* [online]. .NetStudent, 2008. [cit. 24. 2. 2009]. Dostupné z: http://www.netstudent.cz/%C4%8Cl%C3%A1nky/tabid/56/articleType/ArticleView/articleId/144/vod-do-LINQ.aspx.

[19] SCHENKER, G. *The NHibernate FAQ* [online]. 2009. [cit. 1. 3. 2009]. Dostupné z: http://blogs.hibernatingrhinos.com/nhibernate/Default.aspx.

[20] SHARP, J. *Microsoft Visual C# krok za krokem.* Brno : Computer Press, 2006. 528. s. Přel. z: Microsoft Visual C# 2005 Step by Step. ISBN 80-251-1156-3.

[21] STŘÍBNÝ, P. *Představení známého ORM pro .NET* [online]. 2009. [cit. 20. 2. 2009]. Dostupné z: http://stribny.name/zapisnik/?clanky/nhibernate.

[22] ÚOOÚ. *Zákon č. 101/2000 Sb., o ochraně osobních údajů (účinné znění)* [online]. Úřad pro ochranu osobních údajů, 2009. [cit. 17. 4. 2009]. Dostupné z: http://www.uoou.cz/index.php?l=cz&m=left&mid=01:01:00&u1=&u2=&t=.

[23] WIKIPEDIA. *Sandcastle (software)* [online]. Wikipedia, The Free Encyclopedia, 2009. [cit. 2. 5. 2009]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Sandcastle_(software)&oldid=286812087.

[24] WIKIPEDIE. *Bugzilla* [online]. Wikipedie: Otevřená encyklopedie, 2008. [cit. 18. 4. 2009]. Dostupné z: http://cs.wikipedia.org/w/index.php?title=Bugzilla&oldid=3069296.

[25] WIKIPEDIE. *LINQ* [online]. Wikipedie: Otevřená encyklopedie, 2009. [cit. 28. 4. 2009]. Dostupné z: http://cs.wikipedia.org/w/index.php?title=LINQ&oldid=3880789.

[26] WIKIPEDIE. *MONO* [online]. Wikipedie: Otevřená encyklopedie, 2009. [cit. 2. 5. 2009]. Dostupné z: http://cs.wikipedia.org/w/index.php?title=Mono_(platforma)&oldid=3793557.

[27] WIKIPEDIE. *Strojový kód* [online]. Wikipedie: Otevřená encyklopedie, 2009. [cit. 17. 4. 2009]. Dostupné z: http://cs.wikipedia.org/w/index.php?title=Strojov%C3%BD_k%C3%B3d&oldid=3656601.

[28] WIKIPEDIE. *Subversion* [online]. Wikipedie: Otevřená encyklopedie, 2009. [cit. 18. 4. 2009]. Dostupné z: http://cs.wikipedia.org/w/index.php?title=Subversion&oldid=3690765.

[29] WIKIPEDIE. *Unit testing* [online]. Wikipedie: Otevřená encyklopedie, 2009. [cit. 18. 4. 2009]. Dostupné z: http://cs.wikipedia.org/w/index.php?title=Unit_testing&oldid=3771712.

[30] ZLOTSKÝ, O. *Jak spravovat software pomocí Subversion* [online]. Stickfish, s.r.o, 1999-2008. [cit. 20. 7. 2008]. Dostupné z: http://www.abclinuxu.cz/clanky/programovani/jak-spravovat-software-pomoci-subversion-i.

Seznam obrázků

Seznam tabulek

Rejstřík